

基础教程（第2版）



耿祥义 张跃平 编著

清华大学出版社

XML 基础教程（第 2 版）

耿祥义　张跃平　编　著

清华大学出版社
北　京

内 容 简 介

本书主要针对 XML 的初学者，重点放在 XML 中一些重要概念和技术上，所选例题具有较强的针对性，便于理解 XML 中的概念和技术，帮助读者在较短的时间内打下一个良好的基础。全书共分 9 章，内容包括 XML 简介、规范的 XML 文件、有效的 XML 文件、DOM 解析器、SAX 解析器、XPath 语言、XML 与数据库、XML 与 CSS 和 XML Schema 模式简介。

本书适合具有一定 Java 基础和初步 HTML 知识的读者阅读，也适合作为计算机、电子商务、信息类等专业的专业教材和社会培训机构相关专业的培训教材。

图书在版编目(CIP)数据

XML 基础教程/耿祥义，张跃平编著．—2 版．—北京：清华大学出版社，2012.3
ISBN 978-7-302-27782-8

Ⅰ.①X…　Ⅱ.①耿…　②张…　Ⅲ.①可扩充语言，XML—程序设计—教材　Ⅳ.①TP312

中国版本图书馆 CIP 数据核字(2011)第 281280 号

责任编辑：田在儒
封面设计：李　丹
责任校对：李　梅
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　**邮　　编**：100084
社 总 机：010-62770175　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795764
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　**印　张**：12.5　**字　　数**：286 千字
版　　次：2006 年 4 月第 1 版　2012 年 3 月第 2 版　**印　　次**：2012 年 3 月第 1 次印刷
印　　数：1～　000
定　　价：　.00 元

产品编号：043487-01



一、内容结构

本书是《XML 基础教程》的第 2 版，对第 1 版的内容进行了更新，对章节的先后顺序做了调整，使其更加适合教学。全书分为 9 章。第 1 章主要对 XML 做了简单的介绍，帮助读者对 XML 有一个大致的了解。第 2 章详细讲解规范的 XML 文件，使读者认识到规范性的重要作用。第 3 章主要讲解有效的 XML 文件，特别重点讲解 DTD 文件。第 4 章详细讲解 DOM 解析器，特别对如何使用 DOM 生成 XML 文件给予较详细的讲解。第 5 章详细讲解 SAX 解析器，重点体现 SAX 解析器的优势。第 6 章详细讲解 XPath 语言，重点体现 XPath 在检索 XML 文件中数据上的优势。第 7 章讲解 XML 与数据库，使读者领会二者互相转化的重要意义，并掌握如何实现二者转换的设计技术。第 8 章讲解 CSS 技术及如何使用 CSS 显示 XML 中数据的细节。第 9 章介绍 XML Schema 模式，讲解如何用模式约束 XML 标记的数据类型。

二、本书特色

- 强调 XML 基础知识；
- 突出 XML 解析器和 Xpath 查询；
- 结合实例、注重应用。

本书知识点明确，内容衔接流畅、通俗易懂，便于教学和自学。许多例题都是经过精心考虑，所有代码都测试通过。

三、资源下载

读者可登录清华大学出版社网站下载本书全部的例题代码。

四、作者简介

耿祥义，1995 年中国科学技术大学获理学博士学位。1997 年从中山大学博士后流动

站出站。现任大连交通大学教授。已编写出版《Java 设计模式》、《Java 2 实用教程》、《Java 课程设计》、《JSP 实用教程》、《JSP 程序设计》等 10 余部教材。

张跃平,现任大连交通大学讲师。已编写和参编出版《Visual FoxPro 课程设计》、《Java 2 实用教程》、《Java 设计模式》、《JSP 实用教程》和《Java 课程设计》5 部教材。

五、交流沟通

希望本教材能对读者学习 XML 有所帮助,并请读者批评指正(xygeng0629@sina.com)。



第1章 XML简介 …… 1

1.1 什么是XML …… 1
1.2 XML文件的编辑与保存 …… 3
1.3 XML和HTML有何不同 …… 4
1.4 XML解析器 …… 7
1.5 XML的优势 …… 7
习题1 …… 8

第2章 规范的XML文件 …… 10

2.1 规范性 …… 10
2.2 XML声明 …… 11
2.2.1 version属性 …… 11
2.2.2 encoding属性 …… 11
2.2.3 standalone属性 …… 13
2.3 标记 …… 13
2.3.1 空标记 …… 14
2.3.2 非空标记 …… 14
2.3.3 标记的名称 …… 17
2.3.4 根标记 …… 17
2.3.5 标记的子孙关系 …… 17
2.4 特殊字符 …… 18
2.5 CDATA段 …… 18
2.6 标记的文本数据 …… 19

2.7 属性…………………………………………………………………………………… 19
2.7.1 属性的构成 ………………………………………………………………………… 19
2.7.2 使用属性的原则 …………………………………………………………………… 20
2.8 注释…………………………………………………………………………………… 20
2.9 名称空间……………………………………………………………………………… 21
2.9.1 有前缀和无前缀的名称空间 ……………………………………………………… 21
2.9.2 标记中声明名称空间 ……………………………………………………………… 22
2.9.3 名称空间的作用域 ………………………………………………………………… 22
2.9.4 名称空间的名字 …………………………………………………………………… 23
习题 2 ……………………………………………………………………………………… 24

第 3 章 有效的 XML 文件 ………………………………………………………………… 26

3.1 有效 XML 文件的定义 ……………………………………………………………… 27
3.1.1 初识 DTD …………………………………………………………………………… 28
3.1.2 DTD 文件的保存 …………………………………………………………………… 29
3.1.3 XML 文件与 DTD 文件相关联 …………………………………………………… 29
3.2 如何检查有效性……………………………………………………………………… 31
3.3 DTD 中的元素 ……………………………………………………………………… 33
3.3.1 约束标记只包含文本数据 ………………………………………………………… 33
3.3.2 约束标记的子标记 ………………………………………………………………… 34
3.3.3 约束标记的混合内容 ……………………………………………………………… 37
3.3.4 EMPTY 和 ANY …………………………………………………………………… 38
3.4 DTD 的完整性 ……………………………………………………………………… 39
3.5 DTD 中的属性约束列表 …………………………………………………………… 39
3.5.1 ATTLIST 属性约束列表 …………………………………………………………… 40
3.5.2 属性的默认值 ……………………………………………………………………… 42
3.5.3 属性类型 …………………………………………………………………………… 47
3.6 内部 DTD …………………………………………………………………………… 53
习题 3 ……………………………………………………………………………………… 55

第 4 章 DOM 解析器 ……………………………………………………………………… 57

4.1 认识 DOM 解析器 …………………………………………………………………… 57
4.1.1 DOM 标准 ………………………………………………………………………… 57
4.1.2 初识 JAXP ………………………………………………………………………… 58
4.1.3 Document 节点 …………………………………………………………………… 59
4.2 节点的类型…………………………………………………………………………… 61
4.2.1 Node 接口…………………………………………………………………………… 61
4.2.2 Node 接口的常用方法……………………………………………………………… 62

4.2.3 节点的子孙关系 …… 62
4.2.4 使用递归方法输出节点中的数据 …… 62
4.3 Document 节点 …… 64
4.4 Element 节点 …… 64
4.5 Text 节点 …… 67
4.6 Attr 节点 …… 71
4.7 DocumentType 节点 …… 72
4.8 处理空白 …… 74
4.9 验证规范性和有效性 …… 76
4.10 使用 DOM 生成 XML 文件 …… 76
4.10.1 Transformer 对象 …… 77
4.10.2 用于修改 Document 的常用方法 …… 77
4.10.3 用 DOM 建立 XML 文件 …… 78
习题 4 …… 82

第 5 章 SAX 解析器 …… 83

5.1 初识 SAX 解析器 …… 83
5.1.1 SAX 解析器及其工作原理 …… 83
5.1.2 创建 SAX 解析器的步骤与事件处理 …… 84
5.2 文档开始与结束事件 …… 87
5.3 标记开始与结束事件 …… 89
5.4 文本事件 …… 91
5.5 名称空间事件 …… 94
5.6 错误事件 …… 97
5.7 处理空白 …… 101
习题 5 …… 103

第 6 章 XPath 语言 …… 104

6.1 XPath 简介 …… 105
6.1.1 初识 XPath 路径表达式 …… 105
6.1.2 使用 XPath API …… 106
6.2 Node 节点 …… 107
6.2.1 节点之间的关系 …… 107
6.2.2 节点的类型 …… 108
6.2.3 节点的名字与值 …… 110
6.3 XPath 路径表达式的结构 …… 110
6.3.1 绝对路径与相对路径 …… 110
6.3.2 定位步与节点集 …… 111

6.3.3 轴及缩写 …… 114
6.3.4 节点测试 …… 116
6.4 谓词 …… 119
6.4.1 谓词的格式与作用 …… 119
6.4.2 寻找特殊位置的节点 …… 121
6.4.3 寻找有特殊属性的节点 …… 122
6.4.4 寻找有特殊关系节点的节点 …… 124
6.4.5 使用谓词嵌套寻找节点 …… 126
6.5 节点集上使用谓词 …… 127
6.6 节点集的并运算 …… 128
6.7 Java XPath API …… 129
6.7.1 按 NodeSet 计算 …… 129
6.7.2 按 Node 计算 …… 129
6.7.3 按字符串计算 …… 130
6.7.4 按布尔值计算 …… 131
6.7.5 按数值计算 …… 131
6.8 节点集与函数 …… 133
6.9 图书查询 …… 134
习题 6 …… 137

第 7 章 XML 与数据库 …… 141

7.1 JDBC …… 142
7.2 Microsoft Access 数据库 …… 142
7.2.1 建立数据库 …… 142
7.2.2 创建表 …… 143
7.3 连接数据库 …… 143
7.3.1 JDBC-ODBC 桥接器 …… 143
7.3.2 ODBC 数据源 …… 144
7.3.3 建立连接 …… 145
7.4 XML 至数据库 …… 146
7.5 数据库至 XML …… 148
习题 7 …… 151

第 8 章 XML 与 CSS …… 152

8.1 初识 CSS …… 152
8.2 XML 关联 CSS …… 153
8.3 标记与样式表 …… 154
8.3.1 标记的名字与样式表的名称 …… 154

8.3.2 CSS 的显示规则 …… 155
8.4 数据结构与显示相分离 …… 156
8.5 设置文本的显示方式 …… 158
8.5.1 块方式 …… 158
8.5.2 行方式 …… 160
8.5.3 按列表方式 …… 161
8.5.4 不显示 …… 163
8.6 字体 …… 163
8.7 文本样式 …… 165
8.8 边框 …… 167
8.9 边缘 …… 169
8.10 颜色和背景 …… 171
8.11 显示图像 …… 171
8.12 设置鼠标的形状 …… 173
8.13 处理层叠 …… 174
习题 8 …… 175

第 9 章 XML Schema 模式简介 …… 177

9.1 什么是 XML Schema …… 177
9.2 XML Schema 中的标记 …… 178
9.2.1 根标记 …… 178
9.2.2 元素标记 …… 179
9.2.3 属性标记 …… 180
9.3 XML Schema 模式的验证 …… 180
9.4 简单类型元素 …… 184
9.5 复杂类型元素 …… 186
9.6 属性 …… 188
习题 9 …… 190

第 1 章 XML 简介

主要内容

- 什么是 XML
- XML 文件的编辑与保存
- XML 和 HTML 有何不同
- XML 解析器
- XML 的优势

随着网络的迅速发展以及规模的扩大，对信息的规范性提出了更加严格的要求，XML 就是在这一背景下诞生的一种数据格式标准。

在学习这门课程之前读者应当初步了解 HTML，并有一定的 Java 语言基础。另外，需要强调的是，由于许多 Web 技术都和 XML 有关，因此本课程的学习对于进一步学习、理解 Web 技术是非常有帮助的。

1.1 什么是 XML

随着网络的迅速发展，万维网联盟(World Wide Web Consortium，W3C)认识到信息规范化的重要性，即用一种规范化的格式来处理信息，从而可以使得人们更加方便地交互信息。XML 是 eXtensible Markup Language 的缩写，是由万维网联盟定义的一种语言，称为可扩展标记语言。所谓可扩展性是指 XML 允许用户按照 XML 规则自定义标记。

XML 文件是由标记及其所标记的内容构成的文本文件，与 HTML 文件不同的是，这些标记可自由定义，其目的是使得 XML 文件能够很好地体现数据的结构和含义。W3C 推出 XML 的主要目的是使得 Internet 网络上的数据相互交流更方便，让文件的内容更加通俗易懂。

以下是一个简单的 XML 文件。

first.xml

```
<?xml version = "1.0" ?>
<学生>
    <姓名> 张三
      <性别> 男 </性别>
```

```
    <出生日期> 1995/05/15 </出生日期>
</姓名>
<姓名> 翠花
    <性别> 女 </性别>
    <出生日期> 1998/08/08 </出生日期>
  </姓名>
</学生>
```

尽管上述XML文件非常简单,但基本体现了XML文件的基本结构,总结如下(有关细节将在后续章节讲解)。

- XML文件包含一个XML声明,其位置必须在XML文件的首行(有关XML声明将在2.1节详细介绍):

```
<?xml version = "1.0" ?>
```

- XML文件中包含有若干个标记,每个标记由开始标签和结束标签构成。
- XML文件有且仅有一个根标记,其他标记都必须封装在根标记中,文件的标记必须形成树形结构。
- 标记的开始标签和结束标签之间的内容称为该标记所标记的内容,简称标记的内容。一个标记的内容中可以包含文本或其他的标记,其中包含的标记称为该标记的子标记。

上面的XML文件的根标记的开始标签是"＜学生＞",结束标签是"＜/学生＞",该根标记有2个子标记"＜姓名＞…＜/姓名＞"。"＜姓名＞…＜/姓名＞"的内容既有文本也有子标记,例如其中一个标记"＜姓名＞…＜/姓名＞"中的文本是"张三",子标记是"＜性别＞…＜/性别＞"和"＜出生日期＞…＜/出生日期＞"。

XML文件必须符合一定的语法规则(后续的章节会详细地讲解XML的语法规则),只有符合这些规则,XML文件才可以被XML解析器解析,以便利用其中的数据,如图1.1所示。



图1.1 XML解析器

下面的XML文件都是错误的,其中的"errorOne.xml"没有根标记,"errorTwo.xml"虽然有根标记,但根标记的两个"节目"子标记有交叉,导致标记没有形成树形结构。

errorOne.xml

```
<节目> 乡村爱情
  <播出时间> 20时 22分</播出时间>
</节目>
<节目> 借枪
  <播出时间> 22时 38分</播出时间>
```

```
</节目>
```

errorTwo.xml

```
<?xml  version = "1.0"  ?>
<国贸大厦>
   <商品> 电视机
        <价格> 6368 元
   </商品>
        </价格>
   <商品> 手机
      <价格> 2678 元</价格>
   </商品>
</国贸大厦>
```

"商品"与"价格"标签出现交叉

1.2 XML 文件的编辑与保存

1. 编辑与保存

XML 文件是具有特殊扩展名(.xml)的文本文件,因此需使用纯文本编辑器来编辑 XML 文件。本书以 Windows 系统自带的"记事本"做编辑器。XML 文件保存的扩展名必须为".xml",例如"Example.xml"、"hello.xml"等。需要特别注意是,名字区分大小写,hello.xml 和 Hello.xml 是完全不同的 XML 文件的名字。

对于初学者不建议使用集成开发工具来编写 XML 文件,尽管各种集成开发工具也提供了用于编辑 XML 文件的编辑器,但同时也屏蔽了 XML 的许多知识点,这非常不利于学习 XML。在掌握了 XML 知识后,具体开发项目时,应当选择一个流行的、成熟的集成开发工具来编写 XML 文件,这有利于更好、更快地编写 XML 文件,例如 XMLSpy 就是非常优秀的 XML 集成开发工具。对于已掌握 XML 知识的人,学习使用各种集成开发工具是没有任何困难的。

一个 XML 文件应当以 XML 声明作为文件内容的第一行,在其前面不能有空白或其他的处理指令或注释。XML 声明以"＜? xml"标识开始,以"? ＞"标识结束。注意"＜?"和"xml"之间,以及"?"和"＞"之间不能有空格。

以下是一个最基本的 XML 声明:

```
<?xml version = "1.0" ?>
```

一个简单的 XML 声明中可以只包含属性 version,目前该属性的值取 1.0(1.1 还没有正式公布,1.1 增加了一些极少被使用的功能),指出该 XML 文件使用的 XML 版本。XML 声明中还可以指定 encoding 属性的值,该属性规定 XML 文件采用哪种字符集进行编码。例如:

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
```

如果在 XML 声明中没有显示地指定 encoding 属性的值,那么该属性的默认值为 UTF-8 编码。如果 encoding 属性的值为 UTF-8,XML 文件必须按照 UTF-8 编码来保存,这样

XML 解析器就会识别 XML 中的标记并正确解析标记中的内容。

使用文本编辑器“记事本”编辑 XML 文件,在保存文件时,必须将“保存类型”选择为“所有文件”,将“编码”选择为“UTF-8”。如果在保存文件时,系统总是给文件名尾加上“.txt”,那么在保存文件时可以将文件名用双引号括起,如图 1.2 所示。

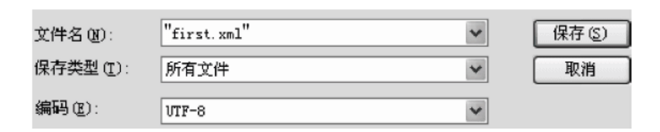


图 1.2　保存 XML 文件

2. 如何检查 XML

XML 的语法规则非常严格,这一点和 HTML 有很大的不同,HTML 本身语法十分不严格,严重影响网络信息传送和共享。W3C 吸取了 HTML 发展的教训,对 XML 指定了严格的语法标准,例如,标记要有一个开始标记和结束标记、所有的标记都必须合理嵌套,即形成树形结构。也就是说 XML 文件必须符合一定的语法规则,只有符合这些规则,XML 文件才可以被 XML 解析器解析,以便利用其中的数据。

XML 文件分为规范的 XML 文件和有效的 XML 文件,符合 W3C 制定的基本规则的 XML 文件称为规范的 XML 文件,规范的 XML 文件如果再符合额外的一些约束就称为有效的 XML 文件。有关 XML 的详细语法将从第 2 章开始讲述。为了检查 XML 文件是否规范,一个简单的办法就是用浏览器,例如 IE6.0。打开 XML 文件,如果 XML 是规范的,浏览器将显示 XML 源文件,否则将显示出错信息。图 1.3 显示了前面第 1.1 节中规范的 XML 文件 first.xml;图 1.4 显示了前面第 1.1 节中不规范的 XML 文件 errorOne.xml。

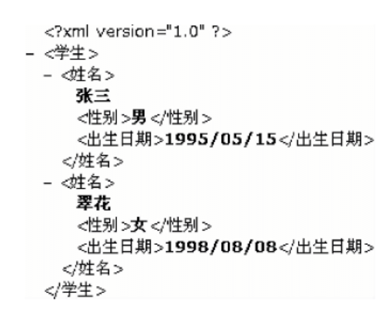


图 1.3　打开规范的 XML 文件

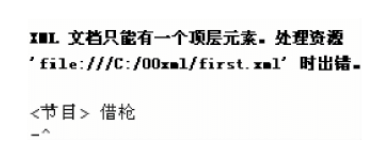


图 1.4　打开不规范的 XML 文件

注意:编写保存一个 XML 文件后,养成用浏览器检查 XML 文件是否有语法错误的习惯。

1.3　XML 和 HTML 有何不同

尽管 XML 和 HTML 都是 W3C 定义的语言,但二者有很大的不同,阐述如下。

1. HTML 的核心

HTML 是用来编写 Web 页的语言(超文本标记语言),HTML 的核心是把数据和数

据的显示外观捆绑在一起，即 HTML 中的标记的出发点不是为了体现数据的含义，而是体现数据的显示格式，整个 HTML 文件的目的不是为了组织数据，而是为了组织某些数据的显示外观。如果某个应用系统只想使用 HTML 中所包含的数据而不需要它的显示外观，可以想象，该应用系统将 HTML 中的数据和外观分离是多么的困难。

另外，HTML 不允许用户自定义标记，目前的 HTML 有一百多个标记。HTML 不能体现数据的组织结构，只能够描述数据的显示格式。下面的 HTML 将数据分别用黑体 1、黑体 2 和黑体 3 来显示。

show. html

```
<html>
    <H1>  张小山
          <H2> 男 </H2>
     </H1>
     <H2>  李翠花
          <H3> 女 </H3>
     </H2>
</html>
```

对于上述 HTML 文件，一个不认识汉字的外国人，并不知道"张小山"和"李翠花"是人的名字。因为 HTML 中的标记的出发点不是为了体现数据的含义，而是为了体现数据的显示格式，并不体现数据的组织结构。浏览器能将 HTML 中标记"＜H1＞…＜/H1＞"，"＜H2＞…＜/H2＞"和"＜H3＞…＜/H3＞"所标记的文本内容分别用黑体 1、黑体 2 和黑体 3 显示在浏览器中，如图 1.5 所示。

张小山
男
李翠花
女

图 1.5 用浏览器打开 HTML 文件

2. XML 的核心

和 HTML 不同的是，XML 的核心是描述数据的组织结构，使 XML 可以作为数据交换的标准格式。XML 可自定义标记，其标记名称是对所标记的数据内容含义的抽象，而不是数据的显示格式，而且 XML 文件通过其中的标记来表示数据的组织结构。例如，下述 XML 文件 second. xml。

second. xml

```
<student>
    <name> 张小山
          <sex> 男 </sex>
     </name>
     <name> 李翠花
          <sex> 女 </sex>
     </name>
</student>
```

对于上述 XML 文件，即使是一个不认识汉字的外国人，也很清楚地知道"张小山"和"李翠花"是学生的名字，并清楚地知道 second. xml 中数据的组织结构。

XML 非常关心数据的组织结构，以便 XML 解析器按照其组织结构分解出数据（有关解析器的详细内容将在 4、5 章讲述），XML 本身不提供数据的显示格式。

XML 有效地分离数据的组织结构和显示外观，即不将显示外观和其中的标记直接进行关联，因此浏览器不能直接显示 XML 文件中的标记的内容。如果需要浏览器显示 XML 文件中标记的内容，就必须以某种方式告诉浏览器如何显示，例如使用层叠样式表(CCS)，如图 1.6 所示(显示 XML 文件中标记的内容的有关知识将在第 8 章讲述)。



图 1.6 XML 文件关联负责显示数据的 CSS

下面的 XML 文件将自己关联到一个层叠样式表，以便浏览器能显示 XML 文件中标记的文本内容。层叠样式表中最重要的组成部分就是样式表，其作用是说明 XML 文件中的标记内容用何种方式来显示。例如，如果想用黑体来显示 XML 文件中标记"<name>…</name>"所标记的文本内容，就可以在层叠样式表文件中包含如下的样式表：

```
name
{  display:block;font - size:36pt; font - style:normal;font - weight:bold
}
```

该样式表的作用是告知浏览器将 XML 文件中标记"<name>…</name>"所标记的文本内容用黑体显示在一个"块区域"中；而

```
sex
{  display:line;font - size:12pt; font - style:italic;
}
```

告知浏览器将 XML 文件中标记"<sex>…</sex >"所标记的文本内容用斜体显示在一行中。

注意：当使用层叠样式表和 XML 关联时，XML 文件中的标记的名字不要含有非 ASCII 码字符(例如，名字中不能有汉字)。

以下的"showXML. css"是一个简单的层叠样式表，该文件可以保存为编码为"ANSI"的文件或编码为"UTF-8"的文件，并和其关联的 XML 文件存放在同一目录中。

showXML. css

```
name
{  display:block;font - size:18pt;font - weight:bold
}
sex
{  display:line;font - size:16pt;font - style:italic
}
birthday
{  display:line;font - size:9pt;font - weight:bold
}
```

XML 文件使用操作指令：

```
<?xml - stylesheet ?>
```

关联到某个层叠样式表，例如：

```
<?xml - stylesheet href = "showXML.css" type = "text/css" ?>
```

以下是一个与 showXML. css 层叠样式表相关联的 XML 文件 three. xml，用浏览器打开 three. xml 文件，浏览器就会按照 showXML. css 中的样式表来显示 XML 文件中标记的文本内容，效果如图 1.7 所示。

three. xml

```
<?xml  version = "1.0"   ?>
<?xml - stylesheet href = "showXML.css" type = "text/css" ?>
< student >
   < name >  张小山
      < sex > 男  </sex >
      < birthday > 1995 年 05 月 15 日 </birthday >
   </name >
   < name >  李翠花
      < sex > 女 </sex >
      < birthday > 1997 年 07 月 27 日 </birthday >
   </name >
</student >
```

张小山 男 1995年05月15日
李翠花 女 1997年07月27日

图 1.7　使用 CSS 显示 XML 中的数据

注意：CSS 是用于布局网页外观的技术，在 XML 中经常会用到 CSS。

1.4　XML 解析器

XML 解析器是 XML 和应用程序之间的一个软件组织，其目的是为应用程序从 XML 文件中解析出所需要的数据。例如，应用程序可能需要 XML 文件所标记的商品价格，并对价格做数据分析处理。现在普遍使用的 XML 解析器都是 Java 语言编写的，本书将采用这样的解析器，有关解析器的详细内容将在 4、5 章讲述。

1.5　XML 的优势

XML 作为表示结构化数据的行业标准，已获得广泛的行业支持。XML 在采用简单、柔性的标准化格式表达以及在应用间交换数据方面是一个革命性的进步。XML 的威力在于不仅提供了直接在数据上工作的通用方法，而且将数据的结构和显示相分离，允许不同来源数据的无缝集成和对同一数据的多种处理。从数据描述语言的角度看，XML 是灵活的、可扩展的，有良好的结构和约束；从数据处理的角度看，它足够简单且易于阅读，几乎和 HTML 一样易于学习，同时又易于被应用程序处理。许多网络应用都在大量使用 XML，XML 已经成为网络应用技术的基础。

通过本书的进一步学习，读者会逐步体会到 XML 的优点。例如，通过 1.3 节的学习已经初步体会到了 XML 是如何将数据的组织结构和显示相分离的，如果想修改 three. xml 的显示外观，只需修改它所关联的 CSS 即可。

习 题 1

1. XML 文件和 HTML 文件有何不同?

2. 如果 XML 文件中的 XML 声明为:

```
<?xml  version = "1.0"   ?>
```

XML 文件应使用怎样的编码保存?

3. 请登录 http://www.w3c.org 网站,了解 XML 的有关话题。

4. 请在搜索引擎中查询 XML Java,看看能查到哪些相关的话题。

5. 下面是一个规范的 XML 文件,阅读该文件并回答提出的问题。

schedule.xml

```
<?xml  version = "1.0"  encoding = "UTF - 8"   ?>
<列车时刻表>
  <车次>
     83 次
    <始发站> 北京 </始发站>
    <终到站> 大连 </终到站>
    <始发时间> 20 时 38 分 </始发时间>
    <终到时间> 07 时 16 分 </终到时间>
  </车次>
</列车时刻表>
```

(1) 是否可以将“＜始发站＞”更改为“＜ 始发站＞”?

(2) 是否可以将“＜始发站＞”更改为“＜始发站 ＞”?

(3) 是否可以将其中的

```
<始发站> 北京 </始发站>
```

更改为:

```
<始发站> 北京 - 大连 </终到站>
```

请用浏览器打开更改后的 schedule.xml 来验证你的结论。

6. 请参考 1.3 节的内容,为下列 XML 文件 time.xml 编写一个显示标记中的文本数据的层叠样式表 showTime.css。

time.xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<?xml - stylesheet href = "showTime.css" type = "text/css" ?>
< root >
    < time >
        北京时间:
        < hour >   12 时   </hour >
        < minute > 56 分   </minute >
```

```
    </time>
    <time>
        格林尼治时间:
        <hour> 4时 </hour>
        <minute>56分</minute>
    </time>
</root>
```

第2章 规范的 XML 文件

主要内容

- 规范性
- XML 声明
- 标记
- CDATA 段
- 标记的文本数据
- 属性
- 注释
- 名称空间

W3C 吸取了当初 HTML 标准不严格的教训，为 XML 制定了严格的语法规则，使得需要 XML 文件的系统很容易开发出解析 XML 文件中相关数据的“XML 解析器”，为信息的相互交流奠定了语法保证。

2.1 规 范 性

按照 W3C 的有关标准，XML 文件分为规范的 XML 文件（Well-Formed XML）和有效的 XML 文件（Validated XML），符合 W3C 制定的基本语法规则的 XML 文件称为规范的 XML 文件，规范的 XML 文件如果再符合额外的一些约束就称为有效的 XML 文件。本章介绍规范的 XML 文件，下一章讲解有效的 XML 文件。

一个规范的 XML 文件应当满足如下语法规则：

- XML 文件用“XML 声明”开始。
- XML 文件有且仅有一个根标记。
- XML 文件的非根标记都必须封装在根标记中。
- 非空标记必须由“开始标签”与“结束标签”构成。
- 空标记没有“开始标签”和“结束标签”。
- XML 文件中的全体标记必须形成树形结构，即标记不允许出现交叉。

图 2.1 示意了一个规范的 XML 文件的基本结构。

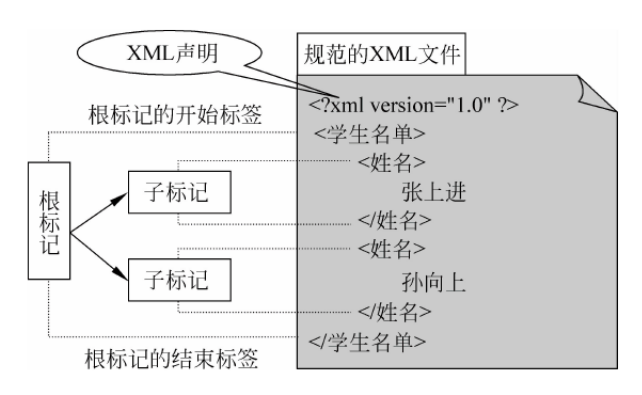


图 2.1 规范的 XML 文件

2.2 XML 声 明

规范的 XML 文件应当以 XML 声明作为文件的第一行，在其前面不能有空白、其他的处理指令或注释。XML 声明以"＜? xml"标识开始，以"? ＞"标识结束。以下是一个最基本的 XML 声明：

```
<?xml version = "1.0" ?>
```

W3C 允许在编写 XML 文件时省略 XML 声明，但是如果一个 XML 文件省略 XML 声明，各种 XML 解析器将默认该 XML 文件是有 XML 声明的，而且 XML 声明是：

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
```

W3C 在 XML 规范中建议每个 XML 文件都显示地写有 XML 声明。

2.2.1 version 属性

一个简单的 XML 声明中可以只包含属性 version，目前该属性只可以取值 1.0，指出该 XML 文件使用的 XML 版本。1.1 版本还没有正式公布，而且仅仅增加了一些极少被使用的功能。如果将 version 属性设置为 1.1，用浏览器(IE6.0)打开 XML 文件时，将得到 XML 版本号设置错误的提示。

2.2.2 encoding 属性

XML 声明中也可以指定 encoding 属性的值，该属性规定 XML 文件采用哪种字符集进行编码。如果在 XML 声明中没有指定 encoding 属性的值，那么该属性的默认值是"UTF-8"。例如：

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
```

声明指定 encoding 属性的值是 UTF-8 编码。如果 XML 使用 UTF-8 编码，那么标记的

名字以及标记包含的文本内容中就可以使用汉字、日文、英文等,XML 解析器就会识别标记的名字并正确解析标记中的文本内容。如果 encoding 属性的值为“UTF-8”,XML 文件必须选择“UTF-8”编码来保存,如图 2.2 所示。

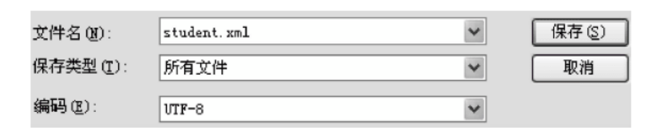


图 2.2 encoding 是 UTF-8 时 XML 文件的保存

如果在编写 XML 文件时只使用 ASCII 字符和汉字,可以将 encoding 属性的值设置为“gb2312”。例如:

```
<?xml  version = "1.0" encoding = "gb2312" ?>
```

这时 XML 文件必须使用“ANSI”编码保存(如图 2.3 所示),XML 解析器根据 encoding 属性的值来识别 XML 文件中的标记并正确解析标记中的文本内容。

文件名(N): goods.xml
保存类型(T): 所有文件
编码(E): ANSI
保存(S)
取消

图 2.3 encoding 是 gb2312 时 XML 文件的保存

如果在编写 XML 文件时只使用 ASCII 字符,可以将 encoding 属性的值设置为“ISO-8859-1”。例如:

```
<?xml version = "1.0" encoding = " ISO - 8859 - 1" ?>
```

这时 XML 文件必须使用“ANSI”编码保存(如图 2.4 所示),XML 解析器根据 encoding 属性的值来识别 XML 文件中的标记并正确解析标记中的文本内容。

文件名(N): temp.xml
保存类型(T): 所有文件
编码(E): ANSI
保存(S)
取消

图 2.4 encoding 是 ISO-8859-1 时 XML 文件的保存

Unicode 字符集由 UNICODE 协会管理并接受其技术上的修改,最多可以识别 65 535 个字符,Unicode 字符集的前 128 个字符刚好是 ASCII 码表。Unicode 字符集还不能覆盖全部历史上的文字,但大部分国家的“字母表”的字母都是 Unicode 字符集中的字符。例如汉字中的“你”字就是 Unicode 字符集中的第 20 320 个字符,一个字符在 Unicode 字符集中的位置也称为该字符的代码点。

XML 文件默认地使用 UTF-8 编码,在 UTF-8 编码中,用一个或几个字节来表示一个字符。UTF-8 编码标准如下:

- 对 Unicode 字符集中代码点 0 到 127 的字符,UTF-8 将该字符编码为 1 个字节,

且高位是 0，也就是说，UTF-8 编码保留 ASCII 字符的编码作为它的一部分。例如，在 UTF-8 和 ASCII 中，"A"的编码都是 0x41(十进制为 65)。

- 对 Unicode 字符集中代码点 128 到 2047 的字符，UTF-8 用 2 个字节来编码，且高字节以"110"作为前缀，低字节以"10"作为前缀。
- 对于 Unicode 字符集中的其他字符，UTF-8 全用 3 个字节来编码，并且这 3 个字节分别用"1110"、"110"和"10"作为前缀。

我们无须知道各种编码的具体细节，只需了解特点即可。例如，对于采用 UTF-8 编码的 XML 文件，在存储它时会多占用一些空间(一个汉字需 3 个字节)，但 UTF-8 编码较好地解决了国际化问题，这一点对于网络上的信息交流是非常重要的，UTF-8 兼容 GB2312、BIG5、EUC-JP 等多种国家的语言的编码。

下面的 Java 应用程序，输出"你"(代码点为 20320)的 UTF-8 编码。

```
class InputUTF_8 {
    public static void main(String args[]) {
        String s1 = "你",s2 = "α";
        try{   byte b[] = s1.getBytes("UTF - 8");
               System.out.print("汉字\'你\'的 UTF - 8 编码: ");
               for(int k = 0;k < b.length;k++) {
                    String str = Integer.toBinaryString(b[k]);
                    str = str.substring(str.length() - 8);
                    System.out.print("   " + str );
               }
        }
        catch(Exception e){}
    }
}
```

上述程序的输出结果是：

```
汉字'你'的 UTF-8 编码:   11100100   10111101   10100000
```

2.2.3 standalone 属性

在 XML 声明中可以指定 standalone 属性的值，该属性的默认值是"no"。该属性可以取值"yes"或"no"，以说明 XML 文件是否是完全自包含的，即是否引用了外部"实体"。下面的 XML 声明指定 standalone 属性的值为"yes"：

```
<?xml version = "1.0" encoding = "UTF - 8" standalone = "yes" ?>
```

2.3 标　　记

XML 文件中的标记分为空标记和非空标记两种。

2.3.1 空标记

1. 语法格式

所谓空标记就是不含有任何内容的标记。由于空标记不含有任何内容,所以空标记不需要开始标签和结束标签,它以“＜”标识开始,用“/＞”标识结束,根据空标记是否含有属性,空标记的语法格式分别为:

```
<空标记的名称  属性列表 />
```

或

```
<空标记的名称 />
```

以下是2个空标记:

```
<张三  age = "28"  sex = "男"  />
    < water />
```

需要注意的是,在标识“＜”和标记名称之间不要含有空格,以下都是错误的空标记:

```
<   张三  age = "28" sex = "男"/>
<   water />
```

在标识“/＞”的前面可以有空格或回行,以下都是正确的空标记:

```
<张三  age = "24" sex = "男"        />
< water     />
```

2. 作用

由于空标记不包含任何内容,因此在实际编写XML文件时,空标记的名称主要用于抽象带有属性的数据,该数据本身并不需要用具体文本进行描述。例如,XML需要描述宽12、长20的长方形,但不准备有任何关于长方形的文字描述,那么就可以使用如下的标记:

```
<长方形  width = "12" length = 20 />
```

XML解析器主要关心空标记中的属性,并可以解析出这些属性的值(见第4、5章)。

2.3.2 非空标记

1. 语法格式

非空标记必须由“开始标签”与“结束标签”构成,“开始标签”与“结束标签”之间是该标记所标记的内容,称为该标记的内容。

开始标签以“＜”标识开始,用“＞”标识结束,“＜”标识与“＞”标识之间是标记的名称和属性列表。根据非空标记是否含有属性,它的语法格式分别为:

```
<标记的名称　属性列表 >
```

或

```
<标记的名称>
```

需要注意的是，在标识“＜”和标记名称之间不能含有空格，允许“＞”的前面有空格或回行。

结束标签以“＜/”标识开始，用“＞”标识结束，“＜/”标识与“＞”标识之间是标记的名称。需要注意的是，在标识“＜/”和标记名称之间不能含有空格，允许“＞”的前面有空格或回行。

在标记的“开始标签”与“结束标签”之间是该标记所包含的内容，以下是一个正确的非空标记：

```
< name> 李云龙 </name >
```

而下面是一个错误的非空标记（“＜”和“name”之间有空格）：

```
<    name> 李云龙 </name >
```

2. 非空标记的内容

在标记的“开始标签”与“结束标签”之间是该标记所包含的内容，一个标记所包含的内容可以由两部分构成：文本数据和标记，其中的标记称为该标记的子标记。

下面的例1说明标记内容中的文本数据和子标记，为了叙述方便，用符号“□”表示编辑操作所输入的空格，“¶”代表编辑操作所输入的回行符。

【例 1】

example2_1.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
< student >¶
    < name >¶
□□□□□□张大山¶
□□□□□□< grade >¶
□□□□□□□□□□一年级¶
□□□□□□</grade >¶
□□</name>¶
</student >
```

上述 example2_1.xml 文件中，标记“name”中的文本数据是：

```
¶
□□□□□□张大山
□□□□□□¶
□□□□□□¶
 □□
```

标记“name”的子标记是“grade”，该子标记“grade”的文本内容是：

```
¶
□□□□□□□□□一年级¶
 □□□□□□
```

请读者注意,以下 3 个标记所包含的文本内容是不同的:

```
<名称>¶
□□□□手机¶
</名称>¶
<名称>手机¶
</名称>¶
<名称>手机</名称>
```

其中,

```
<名称>¶
□□□□手机¶
</名称>
```

包含的文本内容含有 8 个字符,这 8 个字符分别是开始标记后面的 1 个回行符、4 个空格、“手机”和 1 个回行符。

```
<名称>手机¶
</名称>
```

包含的文本内容含有 3 个字符,这 3 个字符分别是开始标记后面的“手机”和 1 个回行符。

```
<名称>手机</名称>
```

包含的文本内容含有 2 个字符,这 2 个字符是开始标记后面的“手机”。

3. 作用

非空标记包含的内容中既可以有文本数据也可以有子标记,当需要用“整体-部分”关系来描述数据时,就可以使用非空标记。例如,“学生”和“姓名”、“学号”之间是“整体-部分”关系,那么就可以让“姓名”、“学号”是“学生”的子标记,以此表示“学生”和“姓名”、“学号”之间是“整体-部分”关系,即 XML 文件中有如下结构的标记:

```
<学生>
   <姓名>张三</姓名>
   <学号>A1001</学号>
</学生>
```

当需要使用文本来描述一个数据时,也需要使用非空标记。例如:

```
<价格>129 元</价格>
<ISBN>7-675-32591-2</ISBN>
```

XML 解析器既关心非空标记包含的子标记也关心它所包含的文本内容,并可以解析出它包含的子标记和文本内容(见 4、5 章)。

需要特别注意的是,下面的标记:

```
<speak></speak>
```

是不包含任何内容的非空标记，或者说是含有“\0”的非空标记（“\0”是空字符），而

```
<speak />
```

才是真正的空标记。

2.3.3　标记的名称

标记的名称必须满足一定的规则，规则是名称可以由字母、数字、下划线（“_”）、点（“.”）或连字符（“-”）组成，但必须以字母或下划线开头。如果 XML 文件使用 UTF-8 编码，字母不仅包括通常的拉丁字母 a、b、c 等，也包括汉字、日文片假名及平假名、朝鲜文以及其他许多语言中的文字。标记名称区分大小写。例如：

```
<name>张三</name>
```

与

```
<Name>张三</Name>
```

是完全不同的标记。

2.3.4　根标记

XML 文件必须有且仅有一个根标记，其他标记都必须封装在根标记中。XML 文件中的全体标记必须形成树形结构。以下是一个不规范的 XML 文件，标记未形成树形结构，“姓名”标记的结束标签与“出生日期”标记的开始标签之间形成交叉。

```
<root>
    <姓名> 张三
    <出生日期>
    </姓名>
      1998 年 12 月 28 日
    </出生日期>
</root>
```

2.3.5　标记的子孙关系

规范的 XML 文件有且仅有一个根标记，其他标记都必须封装在根标记中，文件的标记必须是树形结构。一个标记的子标记的子标记称为该标记的孙标记。称一个标记的子标记为该标记的 1 级子标记，孙标记为该标记的 2 级子标记。本书提到的标记的“子孙”标记泛指该标记的所有级别上的子标记。

2.4 特殊字符

XML 有 5 种字符属于特殊字符，左尖括号“<”、右尖括号“>”、与符号“&”、单引号“'”和双引号“"”。对于这些特殊的字符，XML 有特殊的用途，例如标记的标签中使用左、右尖括号。标记的内容可以由两部分构成：文本数据和标记，按照 W3C 制定的规范，文本数据中不可以含有这些特殊字符。下列标记中的文本内容是非法的：

```
<姓名>  张 & 三 </姓名>
```

要想在文本数据中使用这些特殊字符，办法之一是通过实体引用。XML 有 5 种预定义实体，它们的实体引用格式如下：

- < 引用左尖括号“<”
- > 引用右尖括号“>”
- ' 引用单引号“'”
- " 引用双引号“"”
- & 引用与符号“&”

解析器在解析标记中的数据时，实体引用将被替换为实体引用所引用的实体，例如，“&”被替换为字符“&”。

下列标记中的文本内容是合法的：

```
<姓名>  张 &amp;三 </姓名>
```

解析器解析出该标记的文本数据是：

```
张 & 三
```

2.5 CDATA 段

标记内容中的文本数据不可以含有左尖括号、右尖括号、与符号、单引号和双引号这些特殊字符，如果想使用这些字符，办法之一是通过使用这些字符的引用。但是，如果需要许多这样的字符，文本数据中就会出现很多实体引用或字符引用，导致文本数据的阅读变得困难，CDATA(Character Data)段就是为解决这一问题而引入的。

CDATA 段用“<![CDATA[”作为段的开始，用“]]>”作为段的结束，段开始和段结束之间的内容称为 CDATA 段的内容。解析器不对 CDATA 段的内容做分析处理，因此 CDATA 段中的内容可以包含任意的字符。但是，W3C 规定，CDATA 段中不可以嵌套另一个 CDATA 段。以下是一个正确的 CDATA 段：

```
<![CDATA[
    boolean boo = true&&false
    <你好>
]]>
```

注意：CDATA 段的开始标识“<![CDATA[”以及结束标识“]]>”中不可以有空格字符。不可以把 CDATA 段的开始标识“<![CDATA[”写成“<![cdata[”。

2.6　标记的文本数据

我们已经知道，一个标记包含的内容可以由两部分构成：文本数据部分和子标记部分。一个标记包含的文本数据中可以有普通字符、CDATA 段和所引用的特殊字符（见 2.3 节）。

下面的例 2 中的 example2_2.xml 的根标记“car”的子标记是“audi”，子标记“audi”包含的文本数据是“这是一汽生产的轿车”和一个 CDATA 段。解析器在解析标记中 CDATA 段时不做分析处理，直接获取 CDATA 段中的内容。

【例 2】

example2_2.xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
< car >
< audi > 这是一汽生产的轿车
        <! [ CDATA[
            < power > 2.8 </power >
          ]]>
    </audi >
</car >
```

2.7　属　　性

属性是指标记的属性，可以为标记添加附加信息。

2.7.1　属性的构成

属性是一个“名-值”对，即属性必须由名字和值组成。属性必须在非空标记的开始标签或空标记中声明，用“=”为属性指定一个值。语法如下：

```
<标记名称 属性列表>…</标记名称>
<标记名称 属性列表/>
```

例如：

```
<桌子 width = "300" height = "600" length = "1000"> 办公专用桌 </桌子>
< cloud color = "white" />
```

属性名字的命名规则和标记的命名规则相同，可以由字母、数字、下划线（“_”）、点（“.”）或连字符（“-”）组成，但必须以字母或下划线开头。属性的名字区分大小写。

属性值是一个用单引号或双引号括起的字符串，如果属性值需要包含左尖括号“<”、

右尖括号“＞”、与符号“&”、单引号“'”或双引号“"”,就必须使用字符引用或实体引用。

2.7.2 使用属性的原则

属性不体现数据的结构,只是数据的附加信息。一个信息是否作为一个标记的属性或作为该标记的子标记,这取决于具体的问题。一个基本的原则是:不要因为属性的频繁使用破坏 XML 的数据结构。

下面例 3 中的 XML 文件是一个结构清晰的 XML 文件。

【例 3】

example2_3. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<root>
  <楼房  height = "23m" length = "56m" width = "12m" >
     <结构>
        混土结构
     </结构>
     <建筑商>
        华海建筑集团
     </建筑商>
     <类别>
          商用
     </类别>
  </楼房>
</root>
```

如果把子标记中的数据改为父标记的属性的值,XML 文件的结构就显得很凌乱,如下所示:

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<root>
      <楼房  height = "23m" length = "56m" width = "12m"  结构 = "混土结构"
                  建筑商 = "华海建筑集团"  类别 = "商用" >
      </楼房>
  </root>
```

2.8 注　　释

XML 文件的注释和 HTML 文件相同,以“＜! ——”开始,以“——＞”结束,XML 解析器将忽略注释的内容,不对它们实施解析处理。例如:

```
<?xml version = "1.0"  encoding = "UTF - 8" ?>
<! -- 简单的 XML 文件 -->
<root>
    <speak> 你好 </speak>
```

```
</root>
```

需要注意的是，注释不可以在 XML 声明的前面，下面注释出现的位置是错误的：

```
<!-- 简单的 XML 文件 -->
<?xml version = "1.0" encoding = "UTF-8" ?>
<root>
    <speak> 你好 </speak>
</root>
```

2.9　名称空间

XML 允许自定义标记，那么不同的 XML 文件以及同一 XML 文件中就可能出现名字相同的标记，如果想区分这些标记，就需要使用名称空间。名称空间的目的是有效地区分名字相同的标记，当两个标记的名字相同时，它们可以通过隶属不同的名称空间来相互区分。

在介绍名称空间之前，先看一个简单的例子，下面例 4 的 XML 文件 example2_4.xml 中有两个标记的名字都是“张三”。

【例 4】

example2_4.xml

```
<?xml  version = "1.0"  encoding = "UTF-8" ?>
<简历总汇>
    <张三>  1990 年出生，获得过二等奖学金.</张三>
    <张三>  1991 年出生，曾获得数学竞赛一等奖.</张三>
</简历总汇>
```

在上述 example2_4.xml 文件中有 2 个标记有相同的名字“张三”。如果解析器在解析 XML 文件中的数据时，只想解析出其中一个标记中的数据，就无法通过标记的名称来区分这两个标记。因此当出现名称相同的标记时，如果系统希望给予区分，XML 文件中就应当使用名称空间来区分这样的标记，以便解析器能区分这些名称相同的标记。

XML 通过使用名称空间来区分名字相同的标记，名称空间分为有前缀的名称空间和无前缀的名称空间，以下分别给予讲解。

2.9.1　有前缀和无前缀的名称空间

声明有前缀的名称空间的语法如下：

```
xmlns:前缀 = 名称空间的名字
```

例如：

```
xmlns:person = "China.dalian"
```

声明了一个名字为“China.dalian”的名称空间。

无前缀的名称空间声明语法如下：

```
xmlns = 名称空间的名字
```

例如：

```
xmlns = "www.yahoo.com"
```

注意：声明名称空间时，“xmlns”与“：”以及“：”与名称空间的前缀之间不要有空格。

称两个名称空间相同当且仅当它们的名字相同，即检查声明的名称空间是否相同只需检查所声明的名称空间的名字是否相同即可。也就是说，对于有前缀的名称空间，如果两个名称空间的名字不相同，即使它们的前缀相同，也是不同的名称空间；如果两个名称空间的名字相同，即使它们的前缀不相同，也是相同的名称空间。名称空间的前缀仅仅是为了方便地引用名称空间而已，不能用于区分名称空间是否相同。

下列是三个不同的名称空间：

```
xmlns:north = "liaoning"
xmlns:north = "Liaoning"
xmlns:center = "beijing"
```

下列是两个相同的名称空间(名称为 apple)：

```
xmlns:hello = "apple"
xmlns:ok = "apple"
```

注意：“liaoning”和“Liaoning”是不同的名字，因为名字区分大小写。

2.9.2 标记中声明名称空间

名称空间的声明必须在标记的“开始标签”中，而且名称空间的声明必须放在开始标签中标记名字的后面。例如：

```
<张三 xmlns:p1 = "liaoning">
    1986 年出生.
</张三>
```

2.9.3 名称空间的作用域

标记如果使用了名称空间，那么该名称空间的作用域是该标记及其所有的子孙标记。

如果一个标记中声明的是有前缀的名称空间，那么如果该标记及其子孙标记准备隶属该名称空间，必须通过名称空间的前缀引用这个名称空间，使得该标记隶属于这个名称空间。一个标记通过在标记名字的前面添加名称空间的前缀和冒号来引用名称空间(前缀、冒号和标记名字之间不要有空格)，表明此标记隶属该名称空间。

在下面的例 5 的 example2_5.xml 文件中有两个标记的名字都是“张三”，有两个标记的名字都是“张小三”。通过使用名称空间，让一个“张三”和“张小三”隶属名字为“Liaoning”的名称空间，让另一个“张三”和“张小三”隶属名字为“Beijing”的名称空间。

【例 5】

example2_5.xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<people>
   <p1:张三 xmlns:p1 = "Liaoning">
       1986 年出生,他有一个儿子叫张小三
       <p1:张小三>
           在小学读书
       </p1:张小三>
   </p1:张三>
   <p2:张三 xmlns:p2 = "Beijing">
       1982 年出生,他有一个儿子叫张小三
       <p2:张小三>
           在初中读书
       </p2:张小三>
   </p2:张三>
</people>
```

如果一个标记中声明的是无前缀的名称空间,那么该标记及其子孙标记都默认地隶属于这个名称空间。下面例 6 中的 example2_6.xml 文件中,所有的标记都隶属同一个名字为“http://www.tup.com”的名称空间。

【例 6】

example2_6.xml

```
<book xmlns  = "http://www.tup.com">
   <java>
      Java 基础教程(第 3 版)
      <author>耿祥义 </author>
   </java>
   <jsp>
       JSP 基础教程(第 2 版)
      <author>耿祥义 </author>
   </jsp>
   <xml>
       XML 基础教程(第 2 版)
      <author>耿祥义 </author>
   </xml>
</book>
```

2.9.4　名称空间的名字

名称空间的目的是有效地区分名字相同的标记,那么就涉及如何区分名称空间的名字。W3C 推荐使用统一资源标识符(Uniform Resource Identifier,URI)作为名称空间的名字。URI 是用来标识资源的一个字符串。一个 URI 可以是一个 E-mail 地址、一个文件的绝对路径、一个 Internet 主机的域名等。例如:

```
"http://www.stamp.com"
"c:\\document\\mybook\\java\\hello.txt"
"www.yahoo.com"
```

需要强调的是,在 XML 中,一个 URI 不必是有效的(不必真实地存在这样的资源),XML 使用 URI 仅仅是为了区分名称空间的名字而已。在实践中,大多数 URI 实际上就用统一资源定位符(Uniform Resource Locator,URL)。例如:

```
xmlns = "www.yahoo.com"
xmlns:p = "http://www.yahoo.com"
```

在浏览器的地址栏中输入:

```
www.yahoo.com
```

或

```
http://www.yahoo.com
```

访问的是同一网页。但是,在 XML 中

```
www.yahoo.com
```

和

```
http://www.yahoo.com
```

是完全不同的名称空间,因为二者是不同的字符串。另外,如果在浏览器的地址栏中输入:

```
www.aaabbbccc.com
```

可能会得到"404 File not Found"错误提示,但在 XML 中,"www.aaabbbccc.com"可以作为名称空间的名字。

习　题　2

1. 请阅读下列 XML 文件并回答问题。

```
<?xml  version = "1.0"  encoding = "gb2312" ?>
<root>
   <Keven.J>
     <出生日期>1980.12</出生日期>
     <身高>1.78</身高>
   </Keven.J>
</root>
```

(1) XML 文件应使用什么编码保存?

(2) XML 文件使用 UTF-8 编码保存可以吗?

(3) 将其中的 encoding="gb2312"更改为 encoding="ISO-8859-1"合理吗? 用浏览器打开 XML 文件验证你的结论。

2．标记

```
<出生日期>1980.12</出生日期>
```

和

```
<出生日期>
            1980.12
  </出生日期>
```

所含有的文本内容是否相同？

3．下列哪个是正确的空标记？

(1) ＜a　／＞

(2) ＜a　index＝"97"／＞

(3) ＜　a　／＞

(4) ＜a＞＜／a＞

4．下列哪些 XML 文件是规范的？

A1.xml

```
<root>
  <a> abcd
    <b> 1234
  </a>
        xyz
   </b>
</root>
```

A2.xml

```
<?xml  version = "1.0"   ?>
<root>
  <hello> How are you </Hello>
  <你好> 早上好 </你好 b>
</root>
```

A3.xml

```
<?xml  version = "1.0"   ?>
<root>
  <Hello hi = "How are you"/>
  <Hello></Hello>
  <你好> 早上好 </你好>
</root>
```

5．下面 XML 文件中各个标记包含的文本内容是什么？

```
<?xml  version = "1.0"   ?>
<root>
  <a1 > &lt;CCTV5 &gt; </a1 >
  <a2 >子曰 &quot;有朋自远方来,不亦乐乎 &quot;</a2 >
</root>
```

第 3 章 有效的 XML 文件

主要内容

- 有效 XML 文件的定义
- 如何检查有效性
- DTD 中的元素
- DTD 的完整性
- DTD 中的属性约束列表
- 内部 DTD

XML 的核心是使用标记组织数据结构，以便使信息的交互更加方便。XML 不仅语法严格、语句简练，而且允许自定义标记，使得它很容易被使用。在第 2 章讲述了如何建立一个规范的 XML 文件，规范性仅仅是 XML 语言的基本语法，没有对 XML 文件如何组织数据结构进行具体的约束。如果不对 XML 文件如何组织数据给予一定的约束，那么对同一问题编写的 XML 文件，在数据组织结构上就可能有很大的不同。下面例 1 中有两个关于“商店营业时间”的 XML 文件：time1.xml 和 time2.xml。

【例 1】

time1.xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<商店营业时间>
   <商店>
      <商店名称>国贸大厦</商店名称>
      <营业时间> 08:30 至 18:30 </营业时间>
   </商店>
   <商店>
      <商店名称>华联商场</商店名称>
      <营业时间> 07:30 至 22:30 </营业时间>
   </商店>
</商店营业时间>
```

time2.xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<商店营业时间>
```

```
    <商店>
        <商店名称>国贸大厦</商店名称>
        <开门时间> 08:30 </开门时间>
        <关门时间> 18:30 </关门时间>
    </商店>
    <商店>
        <商店名称>华联商场</商店名称>
        <开门时间> 07:30 </开门时间>
        <关门时间> 22:30 </关门时间>
    </商店>
</商店营业时间>
```

阅读例 1 中的两个 XML 文件，都会知道“国贸大厦”和“华联商场”的营业时间。但是对于解析器，两个文件的数据结构是完全不同的。对于 time1. xml，解析器只要解析出“营业时间”标记中文本数据就可以知道商店的营业时间，而对于 time2. xml，解析器必须分别解析出“开门时间”和“关门时间”标记中的文本数据后，才能知道商店的营业时间。

某些应用系统可能对用 XML 组织数据有特殊要求，例如，管理商店营业时间的系统可能要求在使用 XML 组织商店营业时间相关数据时，将商店的营业开始时间和结束时间分别用不同的标记来描述（如例 1 中的 time2. xml），而不是用一个标记来描述（如例 1 中的 time1. xml）。那么如何检查一个 XML 文件是否符合有关的要求呢？这正是本章要介绍的内容。

3.1 有效 XML 文件的定义

针对某些问题，需要对 XML 文件如何组织数据，即数据结构，进行必要的约束，以便解析器能更好地解析其中的数据。例如，规定某个标记必须有两个子标记，而某个标记不允许有子标记等。对 XML 组织数据进行约束的主要原因有两个，一是使 XML 的数据组织更加合理，符合系统的要求；二是便于维护 XML 中的数据，从而提高整个系统的可维护性。例如，对于例 1 中的两个 XML 文件，管理商店营业时间的系统希望用不同的显示外观来显示开门时间和关门时间，就会要求 XML 文件将营业的开始时间和结束时间分别用不同的标记来描述。

对 XML 的数据结构进行限制有两种方式：使用文档类型定义（Document Type Definition，DTD）和 XML Schema 模式。尽管 DTD 和 XML Schema 模式都可以对 XML 的数据结构进行限制，但二者最重要的区别是 XML Schema 模式是一个特殊的 XML 文件，而 DTD 是有独自语法结构的文件。对 DTD 的使用要早于 XML Schema 模式，DTD 和 XML Schema 模式各有所长，但 XML Schema 模式比 DTD 更为复杂。本章主要讨论 DTD 文件以及如何使用它约束 XML 文件，在第 9 章将讲解 XML Schema 模式。

什么叫一个有效的 XML 文件呢？现在我们就给出正式的定义：

一个规范的 XML 文件如果和某个 DTD 文件相关联，并遵守该 DTD 文件规定的约束条件，就称为有效的 XML 文件。

3.1.1 初识 DTD

在正式涉及 DTD 的细节之前，先通过一个简单的问题介绍如何使用 DTD 文件来约束 XML 文件中的数据结构。

需要一个刻画商店营业时间的 XML 文件，但数据结构必须符合下列要求。

- 根标记的名称是“商店营业时间”。
- 根标记可以有若干个名称为“商店”的子标记。
- 名称为“商店”的标记顺序地包含有名称为“商店名称”、“开门时间”和“关门时间”的子标记。
- 名称为“商店名称”的标记包含的内容只能是文本数据，不能有子标记。
- 名称为“开门时间”的标记包含的内容只能是文本数据，不能有子标记。
- 名称为“关门时间”的标记包含的内容只能是文本数据，不能有子标记。

下面简单介绍 DTD 用怎样的方式来约束一个 XML 文件必须符合上述要求。

DTD 文件是有着特殊结构的文件，简单地说，DTD 文件是由元素所构成的文本文件。在 DTD 文件中，用关键字 ELEMENT 来定义一个元素，格式如下：

```
<!ELEMENT  标记名称  标记的约束条件>
```

在 DTD 文件中，元素的定义是用“＜!ELEMENT”开始，以“＞”结束。要特别注意的是，“＜!ELEMENT”中的“＜”、“!”和“ELEMENT”之间不要有空格字符。

DTD 通过其中的元素来限制 XML 文件中的标记，例如，下面的元素：

```
<!ELEMENT 商店营业时间 (商店*)>
```

可以约束 XML 文件中名称为“商店营业时间”的标记只可以有若干个名字为“商店”的子标记，不可以有任何其他名称的子标记。

下面的元素：

```
<!ELEMENT 商店 (商店名称,开门时间,关门时间)>
```

可以约束 XML 文件中名称为“商店”的标记恰好顺序地有名称分别为“商店名称”、“开门时间”和“关门时间”3 个子标记，而且约束名称为“商店”的标记包含的内容中不可以含有可显示的字符(允许含有空格、回行等空白类字符)。

下列 3 个元素：

```
<!ELEMENT 商店名称 (#PCDATA)>
<!ELEMENT 开门时间 (#PCDATA)>
<!ELEMENT 关门时间 (#PCDATA)>
```

分别约束 XML 文件中名称为“商店名称”、“开门时间”和“关门时间”的标记所包含的内容只可以是文本数据，不可以包含有子标记。

注意：某些书籍将 XML 文件中的标记也称为元素，本书为了符合 HTML，以及后续 Web 设计中的习惯用语，没有在 XML 文件中使用元素这一术语，而是使用了标记这一术

语。需要特别提到的是，DTD 中的元素和 XML 文件中的标记的语法格式有很大的不同。

3.1.2　DTD 文件的保存

DTD 文件是由元素构成的文本文件，需使用文本编辑器编写、保存。DTD 文件的扩展名必须是“.dtd”，在保存 DTD 文件时所选择的编码必须和它要约束的 XML 文件保持一致。例如，DTD 所要约束的 XML 文件的编码为 UTF-8，那么 DTD 文件也必须按照 UTF-8 编码保存。如果在保存文件时，系统总是自动地给文件名尾加上“.txt”，那么在保存文件时可以将文件名字用双引号括起，如图 3.1 所示。

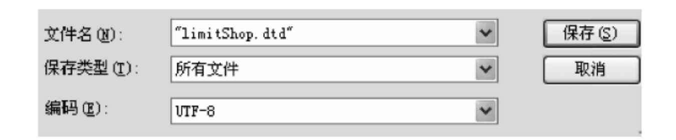


图 3.1　DTD 文件的保存

以下是一个完整的 DTD 文件，可以使用它约束某个 XML 文件(见 3.1.3 小节中的例 2)。

limitShop.dtd

```
<! ELEMENT 商店营业时间 (商店 * ) >
<! ELEMENT 商店 (商店名称,开门时间,关门时间)>
<! ELEMENT 商店名称 (#PCDATA)>
<! ELEMENT 开门时间 (#PCDATA)>
<! ELEMENT 关门时间 (#PCDATA)>
```

3.1.3　XML 文件与 DTD 文件相关联

一个 XML 文件只有和某个 DTD 文件相关联，才会受到该 DTD 文件的约束。如果 XML 文件符合所关联的 DTD 文件中元素所给出的约束，那么这个 XML 文件就是一个有效的 XML 文件。

在 XML 文件中使用“文档类型声明”使当前 XML 文件与一个 DTD 文件相关联。有两种形式的关联：SYSTEM 和 PUBLIC。SYSTEM 关联表明所关联的 DTD 文件由个人或工作小组定义且认可；PUBLIC 关联表明所关联的 DTD 文件已经得到某一领域的认可，是经过许多人讨论得到认可的 DTD 文件。

1. SYSTEM 格式

使用 SYSTEM 文档类型声明的格式：

```
< DOCTYPE 根标记的名称 SYSTEM "DTD 文件的 URI">
```

例如，一个根标记名字是“商店营业时间”的 XML 文件，通过在文件中使用

SYSTEM 文档类型声明：

```
<!DOCTYPE 商店营业时间 SYSTEM "limitShop.dtd">
```

和名字是“limitShop.dtd”的 DTD 文件相关联。

2. PUBLIC 格式

使用 PUBLIC 文档类型声明的格式：

```
<!DOCTYPE 根标记的名称 PUBLIC "正式公用标识符" "DTD 文件的 URI">
```

例如，一个根标记名字是“商店营业时间”的 XML 文件，通过在文件中使用 PUBLIC 文档类型声明：

```
<!DOCTYPE 商店营业时间  PUBLIC  "-//IS0123456/Daxian/ForXML/EN" "limitShop.dtd">
```

和名字是“limitShop.dtd”的 DTD 文件相关联。

在 PUBLIC 文档类型声明中，涉及所谓的“正式公用标识符”(Formal Public Identifier, FPI)，它的格式为：

```
"-ISO 认证//单位名称//DTD 说明//所用语言"
```

需要注意的是，FPI 中不可以含有非 ASCII 码字符(例如含有汉字或日文都是不允许的)。

3. DTD 文件的位置

SYSTEM 或 PUBLIC 格式的文档类型声明中提到的：

```
"DTD 文件的 URI"
```

必须是一个有效的资源，即如果 URI 是一个文件的名字，该 DTD 文件必须和当前 XML 文件在同一目录中；如果 URI 是一个 URL，该 URL 必须是可以访问的。URI 是一个有效的网络资源可以使许多的 XML 文件共享同一个 DTD 文件的约束。例如，URI 是 http://www.data.com/commonFile.dtd(假设这是一个有效的网络资源)，那么就可以使很多 XML 文件共享同一个 DTD 文件的约束。

注意：XML 中使用的与 DTD 相关联的“文档类型声明”一定写在 XML 声明的后面，这是因为 XML 声明必须写在 XML 文件的最前面。

以下例 2 中有一个 XML 文件：example3_2.xml，该 XML 文件和 3.1.2 小节中的 limitShop.dtd 相关联，example3_2.xml 和 limitShop.dtd 保存在相同的目录中(保存时所选择的编码都是 UTF-8)。example3_2.xml 的数据组织结构满足 limitShop.dtd 文件给出的约束条件，即该 XML 文件是有效的 XML 文件。

【例 2】

example3_2.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE 手机用户表 SYSTEM  "limitShop.dtd">
<商店营业时间>
   <商店>
      <商店名称>西单商场</商店名称>
```

```
        <开门时间> 06:30 </开门时间>
        <关门时间> 23:00 </关门时间>
    </商店>
    <商店>
        <商店名称>海淀商厦</商店名称>
        <开门时间> 07:30 </开门时间>
        <关门时间> 18:30 </关门时间>
    </商店>
</商店营业时间>
```

3.2　如何检查有效性

当打开一个 XML 文件时，浏览器的 XML 解析器仅仅检查 XML 文件所关联的 DTD 文件是否有语法错误，并不检查 XML 文件是否遵守该 DTD 规定的约束条件，即浏览器内置的解析器仅仅检查 XML 文件是否是规范的，并不检查 XML 的有效性。

可以使用 DOM 解析器来检查一个 XML 文件是否是有效的，有关 DOM 解析器的内容将在第 4 章详细讲述，以下给出一个简要的步骤说明。

（1）获取一个 DocumentBuilderFactory 对象 factory：

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

（2）factory 设置是否检查 XML 文件的有效性：

```
factory.setValidating(true);
```

（3）factory 对象调用方法 newDocumentBuilder()返回 DOM 解析器：

```
DocumentBuilder domParser =  factory. newDocumentBuilder();
```

（4）解析器 domParser 调用

```
public Document parse(File f) throws SAXException,IOException
```

方法解析 XML 文件。

解析器对象 domParser 有一个成员变量，该变量是 ErrorHandler 类型的对象。domParser 对象在解析文件过程中，每当发现文件有不满足 DTD 文件所给出的约束时，就会通知自己的 ErrorHandler 对象调用相应的方法输出不满足约束的有关信息。需要注意的是，即使 XML 文件不是有效的，也不会影响 parse()方法返回一个 Document 对象，也就是说只要 XML 文件是规范的，DTD 文件本身没有错误，domParser 对象调用 parse()方法终会返回一个 Document 对象。但是，当发生 XML 文件不规范、DTD 文件有语法错误、XML 文件或 DTD 文件不存在等“致命错误”时，解析器将立刻停止解析文件，也就是说 parse()方法不会返回一个 Document 对象。

有关 DOM 解析器将在第 4 章详细讲述，这里不要求看懂解析器的代码。如果读者学习过 Java 语言，只需将下面例 3 中的 TestValidate.java 和要被检查的 XML 文件保存在相同的目录中，然后编译、运行 TestValidate 即可。

图 3.2 是运行 TestValidate 检查 3.1.3 小节例 2 中 example3_2.xml 有效性的效果图。

```
E:\chaper3>java TestValidate
请输入要验证有效性的XML文件的名字：example3_2.xml
example3_2.xml文件是有效的
```

图 3.2 所检查的 XML 文件是有效的

【例 3】

TestValidate.java

```
import javax.xml.parsers.*;
import java.io.*;
import org.w3c.dom.*;
import java.util.Scanner;
public class TestValidate{
   public static void main(String args[]){
       String fileName=null;
       try { Scanner reader=new Scanner(System.in);
               System.out.print("请输入要验证有效性的 XML 文件的名字: ");
               fileName=reader.nextLine();
               DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();
               factory.setValidating(true);
               DocumentBuilder builder= factory.newDocumentBuilder();
               MyHandler handler=new MyHandler();
               builder.setErrorHandler(handler);
               Document document=builder.parse(new File(fileName)) ;
               if(handler.errorMessage==null)
                   System.out.println(fileName+"文件是有效的");
               else
                   System.out.println(fileName+"文件不是有效的");
        }
        catch(Exception e){
               System.out.println(e);
        }
    }
}
class MyHandler extends DefaultHandler{
      String errorMessage=null;
      public void error(SAXParseException e) throws SAXException{
             errorMessage=e.getMessage();
             System.out.println("一般错误: "+errorMessage);
     }
     public void fatalError(SAXParseException e) throws SAXException{
             errorMessage=e.getMessage();
             System.out.println("致命错误: "+errorMessage);
     }
}
```

将例 2 中的 example3_2.xml 的某个名称是“商店”的标记增加一个名称是“地理位置”的子标记，例如：

```
<商店>
   <商店名称>西单商场</商店名称>
   <开门时间>06:30</开门时间>
   <关门时间>23:00</关门时间>
   <地理位置>西单</地理位置>
</商店>
```

将导致 example3_2.xml 不再是有效的 XML 文件，这是因为 DTD 文件约束名字为“商店”的标记恰好顺序地有名称分别为“商店名称”、“开门时间”和“关门时间”3 个子标记，除此之外不能再包含其他子标记。用 TestValidate 检查更改后的 example3_2.xml，解析器指出了 XML 未遵守 DTD 文件给出的约束，效果如图 3.3 所示。

```
一般错误：Element type “地理位置” must be declared.
一般错误：The content of element type “商店” must match “(商店名称,开门时间,关门时间)”.
example3_2.xml文件不是有效的
```

图 3.3　所检查的 XML 文件不是有效的

注意：如果 XML 不是规范的，TestValidate 将直接通知“致命错误”，不再去检查有效性。

3.3　DTD 中的元素

DTD 文件使用元素（ELEMENT）来约束 XML 文件中的标记，在 DTD 文件中使用 ELEMENT 定义一个元素，格式为：

```
<! ELEMENT   标记名称   标记的约束条件 >
```

元素用“＜!ELEMENT”开始，以“＞”结束，要特别注意的是，“＜!ELEMENT”中的“＜”、“!”和“ELEMENT”之间不要有空格字符。例如，元素：

```
<! ELEMENT 学生 (学号,姓名) >
```

约束“学生”标记恰好有“学号”和“姓名”2 个子标记，而且这 2 个子标记在“学生”标记中出现的顺序必须是“学号”、“姓名”，而不是“姓名”、“学号”。

特别需要强调以下两点：

- DTD 文件中的元素的作用与其在 DTD 文件中的书写位置无关。
- DTD 文件中不允许使用多个元素约束同一个标记。

3.3.1　约束标记只包含文本数据

如果约束一个标记没有子标记，也就是说约束标记只能包含文本数据，那么元素格式中的

标记的约束条件

是用小括号括起的#PCDATA,即元素格式如下:

```
<!ELEMENT 标记名称 (#PCDATA)>
```

元素中的关键字“#PCDATA”约束标记只可以包含有文本数据,其文本数据中可以有普通字符、PCDATA段和实体引用。

当元素使用#PCDATA约束一个标记时,该标记所包含的文本数据也可以是一个空字符,甚至该标记可以是一个空标记。例如,对于:

```
<!ELEMENT 姓名 (#PCDATA)>
```

下列3个名称是“姓名”的标记都是符合约束条件的标记:

```
<姓名> 张三 </姓名>
<姓名></姓名>
<姓名 />
```

而

```
<姓名> 李逵
<sex> 男 </sex>
</姓名>
```

是不符合约束条件的标记,其原因是该“姓名”标记包含sex子标记。

3.3.2 约束标记的子标记

1. 元素的格式

若要约束XML文件中某个标记可以有怎样的子标记,例如,标记必须有哪些子标记,子标记是否允许重复出现、出现的顺序如何等,那么DTD中元素格式中的

标记的约束条件

是用小括号括起的子标记列表,即元素格式为:

```
<!ELEMENT 标记名称 (子标记列表)>
```

格式中的

(子标记列表)

是用逗号分隔列出的若干个标记。

我们已经知道,XML文件中的标记可以含有文本数据和子标记,如果在DTD文件中使用具有上述约束条件的元素,这样的元素将约束XML文件中相应的标记只可以有子标记,不可以含有能显示的文本数据,即文本数据仅仅可以由空白类(\t\n\x0B\f\r)字符所组成。

例如,元素:

```
<!ELEMENT 开学时间 (year,month,day)>
```

约束标记"开学时间"恰好有 3 个子标记：year、month 和 day，不可以包含能显示的文本数据，并且这 3 个子标记在"开学时间"标记中出现的顺序必须是 year、month 和 day。

下列"开学时间"标记是符合约束的标记：

```
<开学时间>
   <year> 2010 </year>
   <month> 09 </month>
   <day> 01 </day>
</开学时间>
```

下列"开学时间"标记是不符合约束的标记(原因是子标记的顺序不满足约束条件)：

```
<开学时间>
   <month> 09 </month>
   <year> 2010 <year>
   <day> 01 </day>
```

下列"开学时间"标记也是不符合约束的标记(原因是含有可显示的文本"清华大学")：

```
<开学时间> 清华大学
   <year> 2010 </year>
   <month> 09 </month>
   <day> 01 </day>
</开学时间>
```

2. 模式限定符的使用

元素格式：

```
<!ELEMENT  标记名称  (子标记列表)>
```

中的

```
(子标记列表)
```

的每个子标记的后面可以尾加模式限定符号来约束该子标记出现的次数，不尾加限定符号的子标记必须出现且只能出现一次。限定符号有如下 3 种。

＋：尾加该限定符号的子标记必须出现一次或多次。

＊：尾加该限定符号的子标记可出现零次或多次。

？：尾加该限定符号的子标记可出现零次或一次。

例如：

```
<!ELEMENT 库存商品 (商品名称+,管理员?)>
```

约束标记"库存商品"首先顺序地有几个名字为"商品名称"的子标记，然后再顺序地有零个或一个名字为"管理员"的子标记。

```
(子标记列表)
```

中的子标记也可以是几个标记的“或运算”，而且“或运算”必须用小括号括起。例如，元素：

```
<! ELEMENT 应聘者  (姓名,(本科|硕士|博士),性别 )>
```

约束“应聘者”标记必须顺序地有3个子标记：第一个是“姓名”子标记，第二个是“本科”、“硕士”或“博士”子标记，第三个是“性别”子标记。元素：

```
<! ELEMENT 学生  (姓名,(奖励|处分)*,性别 )>
```

约束“学生”标记顺序地有若干个子标记：第一个是“姓名”子标记，然后是多个(包括零个)“奖励”或“处分”子标记，最后一个是“性别”子标记。

以下例4中的example3_4.xml是一个有效的XML文件，它关联的DTD文件是fourDTD.dtd。例4中的XML文件和DTD文件都按照UTF-8编码保存。

【例4】

fourDTD.dtd

```
<! ELEMENT 应聘信息 (应聘者*)>
<! ELEMENT 应聘者 (姓名,(本科|硕士|博士),(奖励|处分)*,(性别))>
<! ELEMENT 姓名 (#PCDATA)>
<! ELEMENT 本科 (#PCDATA)>
<! ELEMENT 硕士 (#PCDATA)>
<! ELEMENT 博士 (#PCDATA)>
<! ELEMENT 奖励 (#PCDATA)>
<! ELEMENT 处分 (#PCDATA)>
<! ELEMENT 性别 (#PCDATA)>
```

example3_4.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<! DOCTYPE 应聘信息 SYSTEM "fourDTD.dtd">
<应聘信息>
    <应聘者>
        <姓名>张三</姓名>
        <硕士>湖南大学计算机理学硕士</硕士>
        <奖励>获得计算机竞赛一等奖一次</奖励>
        <奖励>获得三次一等奖学金</奖励>
        <处分>被警告处分一次</处分>
        <性别>男</性别>
    </应聘者>
     <应聘者>
        <姓名>李翠花  </姓名>
        <博士>武汉大学自动化专业工学博士</博士>
        <奖励>获得五次一等奖学金</奖励>
        <性别>女</性别>
     </应聘者>
     <应聘者>
        <姓名>王娟娟  </姓名>
        <本科>吉林大学中文系文学学士</本科>
```

```
        <性别> 女 </性别>
    </应聘者>
</应聘信息>
```

3.3.3　约束标记的混合内容

一个标记包含的内容可以由两部分构成：文本数据和标记，其中的标记称为该标记的子标记。在3.3.2小节已经讲述了如何使用DTD约束标记的子标记，例如，元素：

```
<!ELEMENT 开学时间 (year,month,day) >
```

约束标记"开学时间"恰好包含3个子标记year、month和day，不可以包含能显示的文本数据。

如果允许标记的内容既可以包含可显示的文本数据，也可以包含子标记，那么元素格式：

```
<!ELEMENT  标记名称  标记的约束条件 >
```

中的

```
标记的约束条件
```

是关键字"#PCDATA"和若干个子标记的"或运算"，而且该"或运算"必须用小括号括起并尾加一个"*"号，即元素格式如下：

```
<!ELEMENT  标记名称  (#PCDATA|子标记1|子标记2…|子标记m)*>
```

需要特别注意的是，"标记的约束条件"必须用"*"符号结尾，以下给出几种常见的错误写法及错误的原因。

```
(#PCDATA|子标记1|子标记2…|子标记m)
```

是错误的，其原因是未用"*"字符结尾。

```
(#PCDATA|子标记1|子标记2…|子标记m)+
```

是错误的，其原因是使用"+"字符结尾。

```
(#PCDATA|子标记1|子标记2…|子标记m)?
```

是错误的，其原因是使用"?"字符结尾。

另外，"标记的约束条件"也不可以错误地写成：

```
(#PCDATA,子标记1,子标记2,…子标记m)
```

错误原因是"子标记列表"不能有"#PCDATA"分项，例如：

```
<!ELEMENT  T28次  (#PCDATA,hour,minute) >
```

是错误的元素定义。

DTD文件对XML文件中标记包含的混合内容进行约束的元素只能约束标记包含的内容可以有能显示的文本数据和子标记,其中可显示的文本数据可以出现也可以不出现,子标记可以出现零次或多次。约束标记的混合内容的元素的缺点是,只能约束该标记可以有哪些子标记,不能约束这些子标记出现的次数和出现的顺序。另外,约束条件中也不能使用限制符号,下面的写法是错误的:

```
(#PCDATA|子标记1+|子标记2*…|子标记?)
```

约束标记中包含的混合内容的元素相对其他元素的格式要复杂些,而且约束能力有限。一般不提倡标记包含的内容中既有可显示的文本内容又有子标记,其主要原因就是因为不好给出约束条件。

在下面的例5中,DTD文件“fiveDTD.dtd”使用了约束标记的混合内容的元素。例5中的XML文件和DTD文件都按照UTF-8编码保存。

【例5】

fiveDTD.dtd

```
<!ELEMENT 学生列表  (姓名*)>
<!ELEMENT 姓名   (#PCDATA|出生日期|性别)*>
<!ELEMENT 出生日期 (#PCDATA)>
<!ELEMENT 性别 (#PCDATA)>
```

example3_5.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE 学生列表 PUBLIC "-//ISO88//school//ForXML/Ch" "fiveDTD.dtd">
<学生列表>
    <姓名> 张三
      <出生日期> 1993-12-12 </出生日期>
      <性别> 男 </性别>
    </姓名>
    <姓名> 孙翠花
      <出生日期> 1992-01-31 </出生日期>
      <性别> 女 </性别>
    </姓名>
</学生列表>
```

3.3.4 EMPTY和ANY

若要约束一个标记是空标记,或者是只含有空字符的非空标记,元素的“标记的约束条件”可以是关键字EMPTY;若对一个标记没有任何约束,元素的“标记的约束条件”可以是关键字ANY。

例如,对于元素:

```
<!ELEMENT  speak  EMPTY>
```

下列两个标记都是符合约束条件的标记：

```
< speak />
< speak ></speak >
```

对于元素：

```
<! ELEMENT hello ANY >
```

下列两个标记也都是符合约束条件的标记：

```
< hello> how are you </hello>
< hello>
    < english> how are you </english>
    < chinese> 你好</chinese>
</hello>
```

3.4　DTD 的完整性

一个 DTD 文件必须是完整的，即必须满足下列两个条件。

1. 不允许无穷嵌套

如果一个元素约束某个标记可以出现某个子标记，那么对该子标记进行约束的元素就不能约束该标记的子标记是它的父标记。例如，一个 DTD 文件中同时出现下列两个元素是不允许的：

```
<! ELEMENT 学生 (姓名,性别) >
<! ELEMENT 姓名 (学生,拼音) >
```

2. 标记必须有元素对其进行约束

XML 文件中的每个标记都必须在 DTD 中有相应元素对其进行约束，例如下列 DTD 文件 nocomplete. dtd 是不完整的，因为 nocomplete. dtd 中没有约束"开门时间"和"关门时间"标记的元素。

nocomplete. dtd

```
<! ELEMENT 商店营业时间 (商店 * ) >
<! ELEMENT 商店 (商店名称,开门时间,关门时间)>
<! ELEMENT 商店名称 ( # PCDATA)>
```

3.5　DTD 中的属性约束列表

我们已经知道，XML 文件中的标记可以附带属性，标记附带属性的目的是为该标记添加附加信息。标记的属性是一个"名-值"对，即属性必须由名字和值组成。属性必须在非空标记的开始标签或空标记中声明，用"＝"为属性指定一个值。例如，下列名字为"长方体"的空标记有三个属性：

```
<长方体 length = "1000" width = "300" height = "600" />
```

一个信息是否作为一个标记的属性或该标记的子标记,这取决于具体的问题。一个基本的原则是不要因为属性的频繁使用破坏 XML 的数据结构。

3.5.1 ATTLIST 属性约束列表

在 3.4 节讲解了如何使用元素约束 XML 文件中的标记,同样,可以通过在 DTD 文件中定义属性约束列表来约束 XML 文件中的标记的属性。

DTD 使用关键字 ATTLIST 定义一个属性约束列表来约束 XML 文件中标记的属性,在 DTD 中定义属性约束列表的格式为:

```
<! ATTLIST  标记名称
             属性名称 属性类型 默认值情况
             属性名称 属性类型 默认值情况
              ⋮
>
```

属性约束列表用“<!ATTLIST”开始,以“>”结束,用来约束标记中可以有哪些属性以及属性值类型,要特别注意的是“<!ATTLIST”中的“<”、“!”和“ATTLIST”之间不要有空格字符。ATTLIST 定义的属性约束列表被习惯地称为 ATTLIST 属性约束列表。

例如,下列 ATTLIST 属性约束列表:

```
<! ATTLIST 教室
           width  CDATA "0"
           length CDATA "0"
>
```

约束 XML 文件中名称为“教室”的标记有 2 个属性:width 和 length,这 2 个属性的类型是 CDATA,属性的默认值都是字符串“0”。下列 ATTLIST 属性约束列表:

```
<! ATTLIST 姓名  性别  CDATA "男">
```

约束 XML 文件中名称为“姓名”的标记有一个名称为“性别”的属性,该属性的类型是 CDATA,属性的默认值是字符串“男”。

注意:*为了保证 DTD 文件的完整性,标记的每个属性在 DTD 文件中都必须有相应的 ATTLIST 属性约束列表给予约束。*

后面的小节将详细讲解如何使用 ATTLIST 属性约束列表来约束 XML 文件中标记的属性。在进入讨论 ATTLIST 属性约束列表的细节之前,先让我们看一个简单的例子。

下面例 6 中的 XML 文件是有效的 XML 文件,DTD 文件 sixDTD.dtd 使用 ATTLIST 属性约束列表约束 XML 文件中名称为“教室”的标记一定有名称为 length 和 width 的属性。例 6 中的 XML 文件和 DTD 文件按照 UTF-8 编码保存在同一个目录中。

【例 6】

sixDTD. dtd

```
<! ELEMENT 教学楼 (教室 * ) >
<! ELEMENT 教室 (号码,用途) >
<! ELEMENT 号码 ( # PCDATA) >
<! ELEMENT 用途 ( # PCDATA) >
<! ATTLIST 教室 width CDATA "8m">
<! ATTLIST 教室 length CDATA "15m">
```

example3_6. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<! DOCTYPE 教学楼 SYSTEM "sixDTD. dtd">
<教学楼>
    <教室>
        <号码> A101 </号码>
        <用途>自习室</用途>
    </教室>
    <教室 width = "10m" length = "16m">
        <号码> A606 </号码>
        <用途>语音室</用途>
    </教室>
</教学楼>
```

打开 example3_6. xml 文件会发现(如图 3. 4 所示),浏览器显示了“教室”的属性及属性值,其中第一个“教室”标记的 width 和 length 属性的值分别是 8m 和 15m,尽管在编写 example3_6. xml 时没有显示地写出这个“教室”标记的属性,但是浏览器内置的解析器认为它有名字为 width 和 length 的属性,且属性值是 DTD 文件 sixDTD. dtd 中 ATTLIST 属性约束列表给出的默认值。

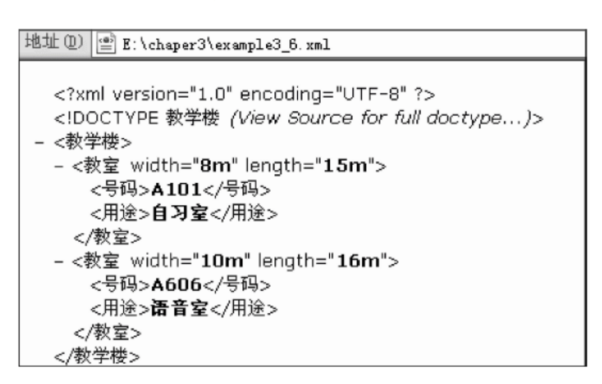


图 3. 4　使用 ATTLIST 属性约束列表约束属性

在 ATTLIST 定义的属性约束列表中涉及“属性名称”、“属性类型”和“默认值情况”,其意义分别阐述如下。

1. 属性名称

属性名称的命名规则和标记的命名规则相同,即属性名称可以由字母(这里所指的字

母是 Unicode 表中的字母,不仅包括通常的拉丁字母,还包括汉字、日文等)、数字、下划线(“_”)、点(“.”)或连字符(“-”)组成,但必须以字母或下划线开头,属性的名称区分大小写。

2. 属性类型

属性值一定是一个字符串,属性类型决定了属性可以用怎样的字符串作为它的值。但是,无论何种类型的属性,其属性值中都不能含有左尖括号“＜”、右尖括号“＞”、与符号“&”、单引号“'”和双引号“"”,如果想使用这些字符,可以使用实体引用。要特别注意的是,属性值区分大小写,TRUE 与 true 是不同的属性值。

3. 默认值情况

ATTLIST 定义的属性约束列表是为了约束 XML 文件中标记的属性,ATTLIST 属性约束列表:

```
<! ATTLIST  标记名称  属性名称 属性类型  默认值情况 >
```

中的“默认值情况”是对标记的属性进行约束的细节条件。后续小节将重点讲解“默认值情况”和“属性类型”的有关细节。

3.5.2 属性的默认值

ATTLIST 属性约束列表:

```
<! ATTLIST  标记名称  属性名称 属性类型  默认值情况 >
```

包括“属性名称”、“属性类型”和“默认值情况”,其中的“默认值情况”含有约束的细节条件。“默认值情况”可以是下列三种情形。

- 字符串

例如:

```
<! ATTLIST  桌子 color CDATA "red">
```

中的“默认值情况”是字符串"red"。

- #IMPLIED 或#REQUIRED

例如:

```
<! ATTLIST 汽车  车牌 CDATA  #REQUIRED>
```

中的“默认值情况”是#REQUIRED。

- #FIXED"字符串"

例如:

```
<! ATTLIST 火警电话  号码 CDATA  #FIXED "119">
```

中的“默认值情况”是#FIXED "119"。

“默认值情况”对标记的属性给予约束的意义如表 3.1 所示。

表 3.1　默认值情况的约束意义

默认值情况	约 束 意 义
字符串	标记必须有该属性，且有默认值
#IMPLIED	标记可以没有该属性，没有默认值
#REQUIRED	标记必须有该属性，没有默认值
#FIXED"字符串"	标记可以没有该属性，但如果有该属性，那么属性值固定不变

以下就“默认值情况”的几种情况进行详细讨论。

1. “默认值情况”是字符串

在 ATTLIST 属性约束列表中可以设置属性的“默认值情况”是一个字符串，例如：

```
<! ATTLIST  桌子 color CDATA "red">
```

如果 ATTLIST 属性约束列表在约束标记的属性时给出的“默认值情况”是一个字符串，那么在编写 XML 时，可以显示地为被约束的标记附加该属性，并且可以重新指定该属性的值。但编写 XML 文件时，也允许被约束的标记不显示地附加该属性，在这种情况下，解析器认为该标记有这个属性，并且默认值为 ATTLIST 属性约束列表中“默认值情况”给定的字符串。

在下面的例 7 中，XML 文件 example3_7. xml 有两个名称是“商品”的标记。其中，第一个“商品”标记显示地附加了 sevenDTD. dtd 文件中 ATTLIST 属性约束列表给出的“类别”属性，并重新指定了该属性的值；第二个“商品”标记没有显示地附加 ATTLIST 属性约束列表给出的“类别”属性，但是浏览器仍然显示了这个标记的属性及属性值，即浏览器内置的解析器认为这个标记有名字为“类别”的属性，且属性值是 DTD 文件 sevenDTD. dtd 中 ATTLIST 属性约束列表中“默认值情况”指定的字符串。

例 7 中的 XML 文件和 DTD 文件按照 UTF-8 编码保存在同一个目录中，用浏览器打开 example3_7. xml 文件的效果如图 3.5 所示。

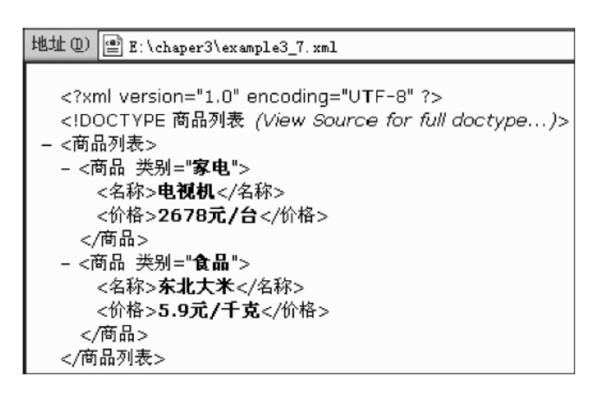


图 3.5　用浏览器显示标记的属性

【例 7】

sevenDTD. dtd

```
<! ELEMENT 商品列表　（商品 * ) >
```

```
<!ELEMENT 商品  (名称,价格)>
<!ELEMENT 名称  (#PCDATA)>
<!ELEMENT 价格  (#PCDATA)>
<!ATTLIST 商品 类别 CDATA "食品">
```

example3_7.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE 商品列表 SYSTEM  "sevenDTD.dtd">
<商品列表>
   <商品 类别="家电">
      <名称>电视机</名称>
      <价格>2678元/台</价格>
   </商品>
   <商品>     <!-- 没有显示地附加"类别"属性 -->
      <名称>东北大米</名称>
      <价格>5.9元/千克</价格>
   </商品>
</商品列表>
```

2. “默认值情况”是#IMPLIED

在ATTLIST属性约束列表中可以设置属性的“默认值情况”是#IMPLIED,例如:

```
<!ATTLIST  桌子 color #IMPLIED>
```

当属性的“默认值情况”是关键字#IMPLIED时,该属性就没有默认值,而且被约束的标记可以不附加该属性。当认为一个属性对于标记可有可无,且没有默认值时,就可以将属性的“默认值情况”设置为#IMPLIED。

3. “默认值情况”是#REQUIRED

在ATTLIST属性约束列表中可以设置属性的“默认值情况”是#REQUIRED,例如:

```
<!ATTLIST  桌子 length #REQUIRED
           桌子 width  #REQUIRED
           桌子 height #REQUIRED
>
```

当属性的“默认值情况”是关键字#REQUIRED时,该属性没有默认值,被约束的标记必须要附加该属性并给出属性的值。当不想为某个属性设置默认值,但要求标记必须附加该属性时,就可以将属性的“默认值情况”设置为#REQUIRED。

4. “默认值情况”是#FIXED"字符串"

在ATTLIST属性约束列表中可以设置属性的“默认值情况”是:

```
#FIXED "字符串"
```

例如:

```
<!ATTLIST 报警电话  号码  CDATA  #FIXED "110">
```

如果属性的“默认值情况”是：

```
#FIXED "字符串"
```

那么该属性的默认值就是关键字#FIXED后面指定的那个字符串。在编写XML文件时，即使被约束的标记不显示地附加该属性，解析器也认为该标记有这个属性，而且属性值是#FIXED后面指定的那个字符串。但是，在编写XML文件时，如果被约束的标记显示地附加该属性，那么不可以改变属性的值，即该属性的值必须是# FIXED后面指定的那个字符串。要约束标记的某个属性的值是固定不变的一个值，就可以将属性的“默认值情况”设置为：

```
#FIXED "字符串"
```

注意：浏览器内置的XML解析器只检查XML关联的DTD文件本身是否有错，不检查XML文件是否遵守了DTD文件的约束条件，建议读者时常使用3.2节中例3提供的TestValiadate解析器验证XML文件是否遵守了DTD文件的约束条件。

在下面的例8中XML文件是有效的，DTD文件约束XML文件中名称为“教室”的标记必须有“号码”属性。例8中的XML文件和DTD文件按照UTF-8编码保存在同一个目录中，用浏览器打开example3_8.xml的效果如图3.6所示。

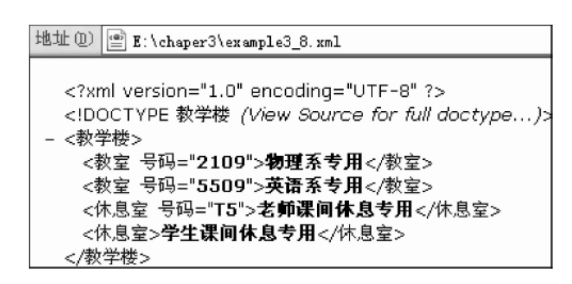


图3.6　“默认值情况”为#REQUIRED和#IMPLIED

【例8】

eightDTD.dtd

```
<!ELEMENT 教学楼  (教室*,休息室*)>
<!ELEMENT 教室 (#PCDATA)>
<!ELEMENT 休息室 (#PCDATA)>
<!ATTLIST 教室   号码 CDATA  #REQUIRED>
<!ATTLIST 休息室   号码 CDATA  #IMPLIED>
```

example3_8.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<!DOCTYPE 教学楼 SYSTEM "eightDTD.dtd">
<教学楼>
    <教室 号码 = "2109"> 物理系专用 </教室>
    <教室 号码 = "5509"> 英语系专用 </教室>
    <休息室 号码 = "T5"> 老师课间休息专用 </休息室>
    <休息室 > 学生课间休息专用 </休息室>
```

```
</教学楼>
```

如果修改 example3_8. xml,使某个“教室”标记不附加属性名字为“号码”的属性,即将其中的

```
<教室 号码 = "2109">物理系专用 </教室>
```

修改为:

```
<教室> 物理系专用 </教室>
```

那么,解析器将检查出这一错误,认为修改后的 example3_8. xml 不是一个有效的 XML 文件,并将错误信息输出。请读者使用 3.2 节中例 3 提供的 TestValiadate 验证修改后的 example3_8. xml。

在下面的例 9 中,DTD 文件约束 XML 文件中的“报警电话”和“火警电话”标记可以附加或不附加“号码”属性。但是如果“报警电话”标记附加“号码”属性,要求属性值必须是“110”;如果“火警电话”标记附加“号码”属性,要求属性值必须是“119”。例 9 中的 XML 文件是有效的,XML 文件和 DTD 文件按照 UTF-8 编码保存在同一个目录中。用浏览器打开 example3_9. xml 的效果如图 3.7 所示。

```
地址(D)  E:\chaper3\example3_9.xml

  <?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE 重要电话簿 (View Source for full doctype...)>
- <重要电话簿>
    <报警电话 号码="110">不要乱打报警电话。</报警电话>
    <火警电话 号码="119">一旦发生火灾，请及时拨打火警电话。</火警电话>
    <急救电话 号码="120">救死扶伤</急救电话>
  </重要电话簿>
```

图 3.7 “默认值情况”为#FIXED

【例 9】

nineDTD. dtd

```
<! ELEMENT 重要电话簿  (报警电话,火警电话,急救电话)>
<! ELEMENT 报警电话 (#PCDATA)>
<! ELEMENT 火警电话 (#PCDATA)>
<! ELEMENT 急救电话 (#PCDATA)>
<! ATTLIST 报警电话   号码 CDATA   #FIXED "110">
<! ATTLIST 火警电话   号码 CDATA   #FIXED "119">
<! ATTLIST 急救电话   号码 CDATA   #FIXED "120">
```

example3_9. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<! DOCTYPE  重要电话簿  SYSTEM  "ten.dtd">
<重要电话簿>
    <报警电话   号码 = "110" > 不要乱打报警电话.</报警电话>
    <火警电话> 一旦发生火灾,请及时拨打火警电话.</火警电话>
    <急救电话> 救死扶伤 </急救电话>
```

```
</重要电话簿>
```

如果修改 example3_9. xml，使“火警电话”标记附加“号码”属性，并指定属性值是“999”，即将其中的

```
<火警电话> 一旦发生火灾,请及时拨打火警电话.</火警电话>
```

修改为：

```
<火警电话  号码 = "999"> 一旦发生火灾,请及时拨打火警电话.</火警电话>
```

那么，解析器将检查出这一错误，认为修改后的 example3_9. xml 不是一个有效的 XML 文件，并将错误信息输出。使用 3.2 节中例 3 提供的 TestValiadate 验证 example3_9. xml 的效果如图 3.8 所示。

```
E:\chaper3>java TestValidate
请输入要验证有效性的XML文件的名字：example3_9.xml
一般错误：Attribute “号码” with value “999” must have a value of “119”.
example3_9.xml文件不是有效的
```

图 3.8 使用解析器检查出的错误

3.5.3 属性类型

属性类型决定了属性可以用怎样的字符串作为属性的值。属性的常用类型有：

CDATA, Enumerated, NMTOKEN, NMTOKENS, ID, IDREF, IDREFS

以下分别来介绍这些属性类型。

1. CDATA 类型

对于 CDATA(Character Data)类型的属性，该属性的值可以是任何一个字符串，但是，字符串中不能含有左尖括号、右尖括号、与符号、单引号和双引号，如果想使用这些字符，可以使用实体引用。

下面的例 10 中，XML 文件是有效的(可以使用 3.2 节中的解析器验证)，DTD 文件的 ATTLIST 属性约束列表约束 XML 文件的标记中的属性是 CDATA 类型，即属性值可以是任何一个字符串。例 10 中的 XML 文件和 DTD 文件按照 UTF-8 编码保存在同一个目录中。

【例 10】

tenDTD. dtd

```
<!ELEMENT 通讯录  (姓名*)>
<!ELEMENT 姓名 (#PCDATA)>
<!ATTLIST 姓名   电话   CDATA   #REQUIRED
                 E-mail CDATA   #REQUIRED
                 住址   CDATA   #IMPLIED
>
```

example3_10. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<! DOCTYPE 通讯录  SYSTEM  "tenDTD.dtd">
<通讯录>
   <姓名 电话 = "12345678" E-mail = "zhangfei@yahoo.com" > 张飞 </姓名>
   <姓名 电话 = "98765432" E-mail = "likui@sohu.com" 住址 = "大连老鳖湾"> 李逵 </姓名>
</通讯录>
```

2. Enumerated 类型

如果属性的类型是 Enumerated 类型,那么该属性只可以是枚举给出的值,即属性可以取的值是用符号“|”分隔的几个字符串中的任何一个。在 ATTLIST 属性约束列表中使用 Enumerated 类型的格式如下:

```
<! ATTLIST 标记名称 属性名称 (属性值 1|属性值 2|…|属性值 n) 默认值情况 >
```

对于 Enumerated 类型,属性值可以是由字母、数字、下划线(“_”)、点(“.”)或连字符(“-”)组成的字符串,并允许首字符是数字字符。需要注意的是,如果 XML 文件使用 UTF-8 编码,字母不仅包括通常的拉丁字母 a、b、c 等,也包括汉字、日文假名、朝鲜文以及其他许多语言中的文字。

下面的例 11 中,XML 文件是有效的,ATTLIST 属性约束列表约束 XML 文件的标记中的属性为 Enumerated 类型,即属性值是从 ATTLIST 属性约束列表中规定的若干个值中选择一个。例 11 中的 XML 文件和 DTD 文件都按照 UTF-8 编码保存在同一个目录中。

【例 11】

elevenDTD. dtd

```
<! ELEMENT 道路  (路灯 * ) >
<! ELEMENT 路灯 (编号,位置)>
<! ELEMENT 编号 ( # PCDATA)>
<! ELEMENT 位置 ( # PCDATA)>
<! ATTLIST 路灯 状态 (亮|灭) # REQUIRED >
```

example3_11. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<! DOCTYPE 道路 SYSTEM "elevenDTD.dtd">
<道路>
   <路灯 状态 = "亮">
      <编号> 10010 </编号>
      <位置> 中山路 </位置>
   </路灯>
   <路灯 状态 = "灭">
      <编号> 20011 </编号>
      <位置> 长江路 </位置>
   </路灯>
</道路>
```

如果将 example3_11. xml 文件中的某个"路灯"标记的"状态"属性的值更改为"turn on",就会导致 example3_11. xml 是无效的("状态"属性的值只能是"亮"或"灭"),用 3.2 节提供的 TestValidate 解析器检查出的错误提示如图 3.9 所示。

```
E:\chaper3>java TestValidate
请输入要验证有效性的XML文件的名字：example3_11.xml
一般错误：Attribute "状态" with value "turn on" must have a value from the list "亮 灭 ".
example3_11.xml文件不是有效的
```

图 3.9　使用解析器检查出的错误

3. NMTOKEN 类型

如果属性的类型是 NMTOKEN,那么属性值可以是由字母、数字、下划线("_")、点(".")或连字符("-")组成的字符串,属性值中不能含有空格字符(属性值也可以用数字、点或连字符开头)。

在下面的例 12 中,XML 文件是有效的,ATTLIST 属性约束列表约束 XML 文件的标记使用 NMTOKEN 类型的属性,即标记附加的属性的属性值中不能含有空格。例 12 中的 XML 文件和 DTD 文件按照 UTF-8 编码保存在同一个目录中。

【例 12】

twelveDTD. dtd

```
<!ELEMENT 作家名单  (姓名 * ) >
<!ELEMENT 姓名 (#PCDATA)>
<!ATTLIST 姓名 笔名 NMTOKEN #IMPLIED >
```

example3_12. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<!DOCTYPE 作家名单 SYSTEM  "twelveDTD.dtd">
<作家名单>
   <姓名 笔名 = "仙山"> 张三 </姓名>
   <姓名 笔名 = "桃李"> 李陶 </姓名>
</作家名单>
```

如果将 example3_12. xml 文件中的

```
<姓名 笔名 = "桃李"> 李陶 </姓名>
```

更改为:

```
<姓名 笔名 = "桃 tao li 李"> 李陶 </姓名>
```

即让属性"笔名"的属性值中包含空格,就会导致 example3_12. xml 是无效的,用 3.2 节提供的 TestValidate 解析器检查出的错误提示如图 3.10 所示。

```
E:\chaper3>java TestValidate
请输入要验证有效性的XML文件的名字：example3_12.xml
一般错误：Attribute value "桃 tao li 李" of type NMTOKEN must be a name token.
example3_12.xml文件不是有效的
```

图 3.10　使用解析器检查出的错误

4. NMTOKENS 类型

NMTOKEN 类型的属性的属性值中不能含有空格。如果需要某个属性的属性值含有空格,而且被空格分隔开的子字符串符合 NMTOKEN 类型属性的属性值规定,那么就可以将属性的类型取为 NMTOKENS 类型(NMTOKENS 关键字比 NMTOKEN 多了一个字母 S,是复数 NMTOKEN)。

在下面的例 13 中,XML 文件是一个有效的 XML 文件,ATTLIST 属性约束列表约束 XML 文件中“图书”标记的“关键字”属性的类型是 NMTOKENS。例 13 中的 XML 文件和 DTD 文件按照 UTF-8 编码保存在同一个目录中。

【例 13】

thirteenDTD.dtd

```
<!ELEMENT 图书列表  (图书*)>
<!ELEMENT 图书 (名称,出版社)>
<!ELEMENT 名称 (#PCDATA)>
<!ELEMENT 出版社 (#PCDATA)>
<!ATTLIST 图书 关键字 NMTOKENS #REQUIRED>
```

example3_13.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE 图书列表 SYSTEM "thirteenDTD.dtd">
<图书列表>
   <图书  关键字="XML 可扩展语言 解析器">
      <名称> XM 基础教程 </名称>
      <出版社> 清华大学出版社 </出版社>
   </图书>
   <图书  关键字="Java 类 对象 线程">
       <名称> Java 程序设计 </名称>
       <出版社> 清华大学出版社 </出版社>
   </图书>
</图书列表>
```

5. ID 类型

ID 类型的属性的属性值具有互斥性,即所有 ID 类型的属性的属性值必须互不相同。如果希望某个属性的属性值具有专用性,即不允许其他 ID 类型的属性与当前属性具有相同的属性值,那么就可以将当前属性的类型取为 ID 类型。需要注意的是,ID 类型的属性值可以由字母、数字、下划线(“_”)、点(“.”)或连字符(“-”)组成,但必须以字母或下划线开头。另外,ID 类型属性的“默认值情况”只能是“#REQUIRED”或“#IMPLIED”,不可以是“字符串”或“#FIXED”。例如,下列“地址”属性的类型是 ID,但“默认值情况”是错误的:

```
<!ATTLIST 汪想 地址 ID FIXED "北京市">
<!ATTLIST 辛望 地址 ID "清华大学">
```

下列“地址”属性的类型是 ID,“默认值情况”是正确的:

```
<! ATTLIST 汪想 地址 ID   #REQUIRED >
<! ATTLIST 辛望 地址 ID   #IMPLIED >
```

在下面的例 14 中，XML 文件是一个有效的 XML 文件，ATTLIST 属性约束列表约束 XML 文件中“名称”标记的“车牌号”属性是 ID 类型。例 14 中的 XML 文件和 DTD 文件按照 UTF-8 编码保存在同一个目录中。

【例 14】

fourteenDTD. dtd

```
<! ELEMENT 城市交通  (公交电车,公交汽车) >
<! ELEMENT 公交电车 (名称 * ) >
<! ELEMENT 公交汽车 (名称 * ) >
<! ELEMENT 名称 ( #PCDATA) >
<! ATTLIST 名称 车牌号 ID #REQUIRED >
```

example3_14. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<! DOCTYPE 城市交通 SYSTEM  "fourteenDTD.dtd">
<城市交通>
  <公交电车>
    <名称 车牌号 = "辽 B555" > 101 无轨电车</名称>
    <名称 车牌号 = "辽 B888" > 201 有轨电车</名称>
  </公交电车>
  <公交汽车>
    <名称 车牌号 = "辽 B666" > 801 快车</名称>
    <名称 车牌号 = "辽 B777" > 631 普通</名称>
  </公交汽车>
</城市交通>
```

如果将 XML 文件 example3_14. xml 中的两个“名称”标记的“车牌号”属性的值都取为“辽 B666”，就会导致 example3_14. xml 是无效的，用 3.2 节提供的 TestValidate 解析器检查出的错误提示如图 3.11 所示。

```
E:\chaper3>java TestValidate
请输入要验证有效性的XML文件的名字：example3_14.xml
一般错误：Attribute value "辽B666" of type ID must be unique within the document.
example3_14.xml文件不是有效的
```

图 3.11 使用解析器检查出的错误

注意：ID 类型属性的属性值的专用性仅限于 ID 类型的属性，并不干涉其他类型属性的属性值。也就是说若干个类型都是 ID 类型的属性的属性值必须互不相同，如果另外一个属性的类型不是 ID 类型，那么它的属性值可以和某个 ID 类型的属性的属性值相同。

一个标记的若干属性中，不允许有两个属性的类型都是 ID。

6. IDREF 类型

如果要通过标记的属性值来判断标记之间的联系，就可以在标记中使用 IDREF

(Identifier Reference)类型的属性。IDREF 类型属性的属性值只能取某个标记中 ID 类型属性的属性值。

下面的例 15 中,XML 文件是一个有效的 XML 文件,其中名称是"专科"、"本科"、"硕士"或"博士"标记的 numberCode 属性是 ID 类型,名字是"姓名"的标记的"学历"属性类型是 IDREF 类型。例 15 中的 XML 文件和 DTD 文件按照 UTF-8 编码保存在同一个目录中。

【例 15】

fifteenDTD. dtd

```
<!ELEMENT 简历列表 (专科,本科,硕士,博士,姓名*)>
<!ELEMENT 专科 EMPTY>
<!ELEMENT 本科 EMPTY>
<!ELEMENT 硕士 EMPTY>
<!ELEMENT 博士 EMPTY>
<!ELEMENT 姓名 (#PCDATA)>
<!ATTLIST 专科 numberCode ID #REQUIRED>
<!ATTLIST 本科 numberCode ID #REQUIRED>
<!ATTLIST 硕士 numberCode ID #REQUIRED>
<!ATTLIST 博士 numberCode ID #REQUIRED>
<!ATTLIST 姓名 学历 IDREF #REQUIRED>
```

example3_15. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<!DOCTYPE 简历列表 SYSTEM "fifteenDTD.dtd">
<简历列表>
   <专科 numberCode = "A101" />
   <本科 numberCode = "B102" />
   <硕士 numberCode = "C201" />
   <博士 numberCode = "D202" />
   <姓名 学历 = "A101"> 张三 </姓名>
   <姓名 学历 = "D202"> 李四 </姓名>
   <姓名 学历 = "B102"> 赵五 </姓名>
   <姓名 学历 = "C201"> 孙六 </姓名>
</简历列表>
```

如果将 example3_15. xml 中的

```
<姓名 学历 = "A101"> 张三 </姓名>
```

更改为:

```
<姓名 学历 = "专科"> 张三 </姓名>
```

就出现不满足 DTD 约束条件的错误,因为"专科"不是 example3_15. xml 中某个 ID 属性的属性值,请读者使用 3. 2 节中提供的 TestValidate 解析器验证修改后的 example3_15. xml 的有效性。

7. IDREFS 类型

若希望约束某个属性的属性值是若干个其他 ID 属性的属性值的组合,IDREFDS 类

型属性的属性值就能满足这一愿望。对于 IDREFS 类型的属性，它的值可以是用空格分隔的若干个其他 ID 属性的属性值。

在下面的例 16 中，ATTLIST 属性约束列表约束 XML 文件中名称是“作者”的标记的“曾编图书”属性的类型是 IDREFS。例 16 中的 XML 文件和 DTD 文件按照 UTF-8 编码保存在同一个目录中。

【例 16】

sixteenDTD.dtd

```
<!ELEMENT 清华大学出版社  (图书*,作者*)>
<!ELEMENT 图书 EMPTY>
<!ATTLIST 图书 ISBN ID #REQUIRED>
<!ELEMENT 作者 (#PCDATA)>
<!ATTLIST 作者 曾编图书 IDREFS #REQUIRED>
```

example3_16.xml

```
<?xml  version="1.0"  encoding="UTF-8" ?>
<!DOCTYPE 清华大学出版社  SYSTEM  "sixteenDTD.dtd">
<清华大学出版社>
    <图书  ISBN="a1111" />
    <图书  ISBN="b2222" />
    <图书  ISBN="c5555" />
    <图书  ISBN="d6666" />
    <作者 曾编图书="a1111  b2222"> 张三 </作者>
    <作者 曾编图书="c5555  d6666"> 李四 </作者>
</清华大学出版社>
```

3.6　内部 DTD

可以把 DTD 文件的内容直接写在 XML 文件的内部，相对外部 DTD 文件，这样的内容称为 XML 文件的内部 DTD。

在 XML 文件中，内部 DTD 用“<!DOCTYPE 根标记名称 [”开始，以“]>”结束。“<!DOCTYPE 根标记名称 [”与“]>”之间是 DTD 的内容。内部 DTD 要写在 XML 文件的“XML 声明”和根标记的“开始标签”之间。

一个 XML 文件如果遵守内部 DTD 规定的约束条件，也称为有效的 XML 文件。

下面的例 17 是一个带有内部 DTD 的 XML 文件。

【例 17】

example3_17.xml

```
<?xml  version="1.0"  encoding="UTF-8" ?>
<!DOCTYPE 学生名单 [
  <!ELEMENT 学生名单 (学生*)>
  <!ELEMENT 学生 (姓名,学号)>
```

```
    <!ELEMENT 姓名 (#PCDATA)>
    <!ELEMENT 学号 (#PCDATA)>
]>
<学生名单>
    <学生>
        <姓名> 张三 </姓名>
        <学号> 2010 </学号>
    </学生>
    <学生>
        <姓名> 李四 </姓名>
        <学号> 2012 </学号>
    </学生>
</学生名单>
```

如果XML文件同时带有内部DTD和外部DTD文件,那么XML解析器会将二者合一。如果内部的DTD和外部的DTD文件同时使用元素限制了某个标记,就会导致错误,因为,不能使用多个元素约束同一个标记。

如果不更改一个已经得到广泛认可的DTD文件,但需增加新的标记及相应的约束条件,就可以同时使用内部DTD和外部DTD。内部DTD可以在不改变外部DTD文件的情况下,方便地增加新的约束条件,便于系统的维护。

同时使用外部DTD和内部DTD的格式如下:

```
<!DOCTYPE 根标记名字 SYSTEM  "外部 DTD 的 URI"  [
    内部 DTD 内容
]>
```

或

```
<!DOCTYPE 根标记名字 PUBLIC  "正式公用标识符" "外部 DTD 的 URI"  [
    内部 DTD 内容
]>
```

假如已经有如下的XML文件:

```
<楼房>
    <办公室>
        <门>木门</门>
        <窗>铝合金</窗>
    </办公室>
</楼房>
```

以及相应的对其进行约束的DTD文件:

```
<!ELEMENT 楼房  ANY>
<!ELEMENT 办公室 (门,窗)>
<!ELEMENT 门 (#PCDATA)>
<!ELEMENT 窗 (#PCDATA)>
```

如果准备修改上述XML文件,增加一个“库房”标记,但不想修改DTD文件,就可以在XML文件中使用内部DTD给出新增标记的约束条件。

在下面的例 18 中，XML 文件同时使用外部 DTD 和内部 DTD 的 XML 文件，使用内部 DTD 给出“库房”标记的约束条件。

【例 18】

eighteenDTD.dtd

```
<!ELEMENT 楼房   ANY >
<!ELEMENT 办公室 (门,窗) >
<!ELEMENT 门 (#PCDATA) >
<!ELEMENT 窗 (#PCDATA) >
```

example3_18.xml

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!DOCTYPE 楼房 SYSTEM "twentyOneDTD.dtd"   [
    <!ELEMENT 库房 (门,窗) >
]>
<楼房>
     <办公室>
         <门>木门</门>
         <窗>铝合金</窗>
     </办公室>
     <库房>
         <门>铁门</门>
         <窗>铝合金</窗>
     </库房>
</楼房>
```

习 题 3

1. 什么叫有效的 XML 文件？

2. DTD 文件的编码必须和其约束的 XML 文件的编码相一致吗？

3. DTD 文件中元素的作用与其在 DTD 文件中的书写位置有关吗？

4. XML 文件如何和一个 DTD 文件关联？

5. 对于一个有效的 XML 文件，标记中的属性一定要有 ATTLIST 属性约束列表对其进行约束吗？

6. 下列 ATTLIST 属性约束列表有何不同？

(1) `<!ATTLIST   张三   学号 CDATA   #REQUIRED >`

(2) `<!ATTLIST   张三   学号 CDATA   #FIXED   "220123">`

(3) `<!ATTLIST   张三   学号 CDATA   ##IMPLIED >`

7. ID 类型的属性有什么特点？

8. 若 XML 文件中没有标记的属性是 ID 类型，那么将某个属性的类型约束为 IDREF 类型是否合理？

9. 如果一个属性的类型是 NMTOKEN，下列哪个字符串是该属性可以取的属性值？

(1) hello

(2) How are you

(3) _Good

(4) 2002-12-22

10. 若有如下的DTD文件:

A. dtd

```
<! ELEMENT 成绩单 (学生 * ) >
<! ELEMENT 学生 (姓名,成绩) >
<! ELEMENT 姓名 ( # PCDATA)>
<! ELEMENT 成绩 ( # PCDATA)>
```

请问,下面的XML文件是有效的吗?如果不是有效的,请将其修改为有效的。

B. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<! DOCTYPE 成绩单 SYSTEM "A.dtd">
<成绩单>
    <学生>
      <姓名> 张三 </姓名>
      <成绩> 优秀 </成绩>
    </学生>
    <学生>
      <成绩> 良好 </成绩>
      <姓名> 李四 </姓名>
    </学生>
</成绩单>
```

第4章 DOM 解析器

主要内容

- 认识 DOM 解析器
- 节点的类型
- Document 节点
- Element 节点
- Text 节点
- Attr 节点
- DocumentType 节点
- 处理空白
- 验证规范性和有效性
- 使用 DOM 生成 XML 文件

针对 XML 文件的解析器是 XML 文件和应用程序之间的一个软件组织，使应用程序从 XML 文件中解析出所需要的数据。有两种最常用的解析器：基于 DOM 的解析器和基于事件的解析器，本章讲述基于 DOM 的解析器，下一章讲述基于事件的解析器(SAX)。

4.1 认识 DOM 解析器

DOM 解析器的核心是在内存中建立和 XML 文件相对应的树形结构数据，XML 文件的标记及其文本内容等都会和内存中树形结构数据的某个节点相对应。使用 DOM 解析器的好处是：一个应用程序可以方便地操作内存中树形结构数据的节点来处理 XML 文件，以便获取所需要的数据。

4.1.1 DOM 标准

DOM(Document Object Model，文档对象模型)是 W3C 制定的一套规范标准，各种基于 DOM 规范的解析器必须按照 DOM 规范在内存中建立数据。DOM 规范的核心是树模型，作为解析 XML 文件的解析器，解析器通过读入 XML 文件在内存中建立一个树

形结构数据，也就是说 XML 文件的标记、标记的文本内容等都会和内存中树形结构数据的某个节点相对应。一个应用程序可以方便地操作内存中树形结构数据的节点来处理 XML 文件，获取所需要的数据。

4.1.2 初识 JAXP

本章主要介绍 Java 语言的 DOM 解析器，Java 语言的 DOM 解析器是支持 DOM level 3 的解析器。按照 W3C 制定的 DOM 规范，Sun 公司发布的 JDK1.4 的后续版本中提供了解析 XML 文件的 JAXP(Java API for XML Parsing，JAXP)，JAXP 实现了 DOM 规范的 Java 语言绑定。可以登录 Sun 公司的网站：http://java.sun.com，免费下载 JDK1.6(例如：jdk-6u3-windows-i586-p.exe)。

首先让我们简单地了解一下 JAXP，有关细节将在后续小节分别讲述。

在 JAXP 中，DOM 解析器是 DocumentBuilder 类的一个实例，该实例由 DocumentBuilderFactory 负责创建，步骤如下。

(1) 使用 javax.xml.parsers 包中的 DocumentBuilderFactory 类调用其类方法 newInstance()实例化一个 DocumentBuilderFactory 对象，例如：

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

(2) 将步骤(1) 得到的 DocumentBuilderFactory 对象调用 newDocumentBuilder()方法返回一个 DocumentBuilder 对象(DocumentBuilder 类在 javax.xml.parsers 包中)，该对象被称为 DOM 解析器，例如：

```
DocumentBuilder domParse = factory.newDocumentBuilder();
```

(3) 将步骤(2)中得到的 DOM 解析器调用 public Document parse(File f)方法解析参数 f 指定的 XML 文件，该方法返回的对象是实现 Document 接口的一个实例(Document 接口在 org.w3c.dom 包中)，例如：

```
Document document = domParse.parse(new File("student.xml")) ;
```

在上面的第(3) 步中，DOM 解析器 domParse 调用 parse() 方法返回的对象：document。它是由一些 Node 对象所构成的，Node 对象被习惯地称为节点，这些节点组成树形结构。document 的结构和 XML 标记组成的树形结构相同，即 document 就是 DOM 解析器在内存中建立的和 XML 文件相对应的树形结构数据。

应用程序分析内存中的树形结构数据 document，就可获得 XML 文件中的各种数据了。

另外，DOM 解析器 domParse 也可以使用下述两个方法解析 XML 文件：

```
public Document parse(InputStream in) throws SAXException, IOException
public Document parse(String uri) throws SAXException, IOException
```

方法 parse(InputStream in)可以解析输入流参数 in 指向的 XML 文件，例如：

```
FileInputStream in = new FileInputStream("price.xml");
```

```
Document document = domParse.parse(in);
```

方法 parse(String uri)可以解析参数 uri 指定的一个有效的资源,如果 uri 是一个 URL,该 URL 必须是可以访问的,例如:

```
String uri = "http://192.168.2.1/price.xml";
Document document = builder.parse(uri);
```

4.1.3 Document 节点

DOM 解析器的 parse()方法把被解析的 XML 文件封装成一个 Document 节点返回[XML 文件和内存中的 Document 节点相对应,见 4.1.2 小节的步骤(3)]。因此,称 Document 对象为 Document 节点。应用程序可以从 Document 节点的子孙节点中获取 XML 文件的各个标记包含的数据的细节。

Document 节点是"树"的根节点,XML 文件中的标记都和 Document 节点的某个子节点相对应。Element 类和 Text 类是比较重要的两个类,这两个类的对象分别称为 Document 节点的 Element 类型子节点(简称 Element 节点)和 Text 类型子节点(简称 Text 节点)。一个 Element 类型节点中还可含有 Element 类型子节点、Text 类型子节点(有关节点类型的细节在下节讲述)。

XML 文件和 Document 节点相对应。XML 文件的根标记和 Document 节点的一个 Element 类型子节点相对应。所以 Document 节点调用 getDocumentElement()方法将返回 XML 文件的根标记所对应的这个 Element 类型子节点。

XML 文件的其他标记都必须封装在根标记中,因此 XML 文件中根标记的某级别上的子标记恰好对应根标记所对应的 Element 节点的同级别上的某个 Element 子节点。

我们已经知道,XML 文件中的标记除了可以含有子标记外,还可以含有文本。如果 XML 文件的某个标记既含有子标记也含有文本,那么该标记在 Document 节点中所对应的 Element 子节点就会有 Element 子节点和 Text 子节点。

注意:Document 节点、Element 节点和 Text 节点都是 Node 节点中的节点。

让我们看一个简单的例子,下面例 1 中的 XML 文件 example4_1.xml 对应的 Document 节点如图 4.1 所示。读者需要把例 1 中的 JAXPOne.java 和它要解析的 example4_1.xml 保存在相同的目录中,然后编译、运行例 1 中的 Java 程序,运行效果如图 4.2 所示。

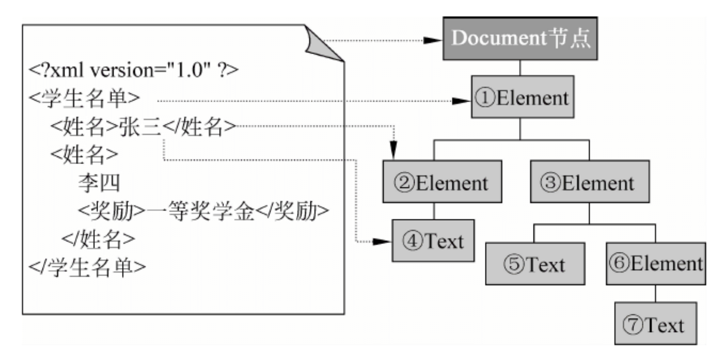


图 4.1 example4_1.xml 文件对应的 Document 节点

【例 1】

example4_1.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<学生名单>
  <姓名>张三</姓名>
  <姓名>
     李四
     <奖励>一等奖学金</奖励>
  </姓名>
</学生名单>
```

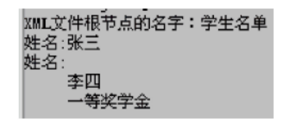


图 4.2　使用解析器解析 XML 文件

JAXPOne.java

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
public class JAXPOne{
   public static void main(String args[]){
      try { DocumentBuilderFactory factory =
               DocumentBuilderFactory.newInstance();
               DocumentBuilder domPaser = factory.newDocumentBuilder();
               Document document = domPaser.parse(new File("example4_1.xml"));
               Element root = document.getDocumentElement();
               String rootName = root.getNodeName();
               System.out.println("XML 文件根节点的名字: " + rootName);
               NodeList nodelist = root.getElementsByTagName("姓名");
               int size = nodelist.getLength();
               for(int k = 0;k < size;k++){
                     Node node = nodelist.item(k);
                     String name = node.getNodeName();
                     String content = node.getTextContent();
                     System.out.print(name);
                     System.out.println(":" + content);
               }
         }
         catch(Exception e){
               System.out.println(e);
         }
   }
}
```

现在,结合前面的图 4.1 来说明 XML 文件和内存中的 Document 节点的对应关系,为了方便,在图 4.1 中使用了①②…数字记号。

- XML 文件中的根标记对应着数字记号为①的 Element 节点。
- XML 文件中标记名称为“姓名”的两个标记按顺序分别对应着 Document 节点中数字记号为②、③的 Element 节点。
- 其中一个“姓名”标记所包含的文本内容“张三”对应着 Document 节点中数字记号为④的 Text 节点。

- 其中一个“姓名”标记所包含的文本内容“李四”对应着 Document 节点中数字记号为⑤的 Text 节点；名字为“奖励”的子标记对应着 Document 节点中数字记号为⑥的 Element 节点；“奖励”子标记所包含的文本内容“一等奖学金”对应着 Document 节点中数字记号为⑦的 Text 节点。

以下分析 JAXPOne.java 中的代码。

Document 节点调用 getDocumentElement()方法将返回 XML 文件的根标记对应的 Element 节点：

```
Element root = document.getDocumentElement();
```

代码：

```
NodeList nodelist = root.getElementsByTagName("姓名");
```

返回一个 NodeList 对象，因为 XML 文件的根标记有 2 个名字为“姓名”的子标记，因此代码：

```
int size = nodelist.getLength();
```

中 size 的值是 2，即 nodelist 刚好含有 2 个 Node 节点。nodelist 中的 Node 节点调用 getTextContent()方法可以返回和该 Node 节点相对应 XML 标记及其所有子孙标记中的文本内容。

注意：DOM 解析器的关键点是得到一个 Document 节点，该节点就是内存中和 XML 文件对应的树形结构数据，即它由若干个子节点构成，这些子节点与 Document 节点形成树形结构，该树形结构的根就是这个 Document 节点。

4.2　节点的类型

4.2.1　Node 接口

Document 接口也是 Node 接口的子接口，也就是说，parse()方法将整个被解析的 XML 文件封装成一个节点返回(XML 文件和内存中的 Document 节点相对应)，并且该节点和它的子节点组成树形结构。因此，应用程序可以从 Document 节点的子孙节点中获取整个 XML 文件中数据的细节(见 4.1 节中的图 4.1)。

按照 DOM 规范，下列接口都是 Node 接口的常用子接口：

- Document 接口负责对应整个 XML 文件。
- Element 接口负责对应标记。
- Text 接口负责对应标记所包含的文本数据。
- Attr 接口负责对应标记的属性。
- DocumentType 接口负责对应 XML 文件所关联的 DTD 文件。

任何实现上述某个接口的类的实例都称为一个节点，例如实现 Element 接口的实例

称为 Element 类型节点,简称 Element 节点。

4.2.2 Node 接口的常用方法

- short getNodeType() 节点调用该方法返回一个表示节点类型的常量(Node 接口规定的常量值),例如,如果节点是 Element 节点,那么节点调用 getNodeType()方法返回的值为 Node.ELEMENT_NODE。
- NodeList getChildNodes() 节点调用该方法返回一个由当前节点的所有子节点组成的 NodeList 对象。
- Node getFirstChild() 节点调用该方法返回当前节点的第一个子节点。
- Node getLastChild() 节点调用该方法返回当前节点的最后一个子节点。
- NodeList getTextContent()节点调用该方法返回当前节点及其所有子孙节点中的文本内容。

4.2.3 节点的子孙关系

为了解析规范的 XML 文件,DOM 规范规定了各种类型节点之间可以形成的子孙关系。例如,Document 节点有且仅有一个 Element 节点,也可以有一个 DocumentType 节点(规范的 XML 文件有且仅有一个根标记,也可以有一个与其关联的 DTD 文件),Element 节点可以有 Element 子节点、Text 子节点(规范的 XML 文件中的标记可以有子标记、文本)。即 Document 节点、Element 节点可以有子节点,但 Text 节点不能再有子节点(属于叶节点)。图 4.3 示意了 DOM 规范所规定的节点之间可以形成的子孙关系。

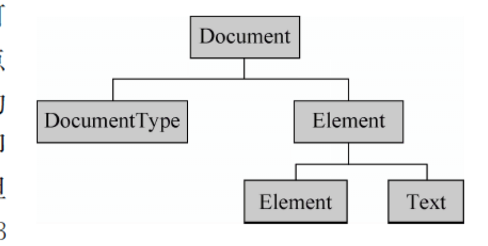


图 4.3 节点可以形成的子孙关系

通过图 4.3 可知,JAXP 中 DOM 解析器的 parse()方法返回的刚好是 DOM 规范规定的根节点 Document,其他类型节点都是此根节点的子孙节点。从 4.3 节开始,将详细地介绍这些子孙节点。

4.2.4 使用递归方法输出节点中的数据

节点调用 getNodeType()方法返回一个表示节点类型的常量(Node 接口规定的常量值),例如,节点是 Element 节点,那么节点调用 getNodeType()方法返回的值为 Node.ELEMENT_NODE。因此可以通过判断节点的类型来输出和节点相关的数据,例如当节点类型是 Element 节点时,就输出节点的名字,当节点是 Text 节点时就输出节点中的数据等。下面的例 2,JAXPTwo.java 使用递归方法输出节点的名称以及节点中的数据。例 2 中 Java 程序的运行效果如图 4.4 所示。

【例 2】

example4_2.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<雇员列表>
    <雇员>
        <姓名> 张三</姓名>
        <年龄> 25 岁</年龄>
        <工资> 3190 元/月</工资>
    </雇员>
    <雇员>
        <姓名> 李四</姓名>
        <年龄> 35 岁</年龄>
        <工资> 4320 元/月</工资>
    </雇员>
</雇员列表>
```

```
雇员列表:
    雇员:
        姓名: 张三
        年龄: 25岁
        工资: 3190元/月

    雇员:
        姓名: 李四
        年龄: 35岁
        工资: 4320元/月
```

图 4.4 输出 XML 中的数据

JAXPTwo.java

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
public class JAXPTwo{
  public static void main(String args[ ]){
     try{  DocumentBuilderFactory factory =
           DocumentBuilderFactory.newInstance();
           DocumentBuilder domParser =  factory.newDocumentBuilder();
           Document document = domParser.parse(new File("example4_2.xml"));
           NodeList nodeList = document.getChildNodes();
           output(nodeList);
     }
     catch(Exception e){
           System.out.println(e);
     }
  }
  public static void output(NodeList nodeList){   //output 是一个递归方法
     int size = nodeList.getLength();
     for(int k = 0;k < size;k++){
        Node node = nodeList.item(k);
        if(node.getNodeType() == Node.TEXT_NODE){
            Text textNode = (Text)node;
            String content = textNode.getWholeText();
            System.out.print(content);
        }
        if(node.getNodeType() == Node.ELEMENT_NODE){
            Element elementNode = (Element)node;
            String name = elementNode.getNodeName();
            System.out.print(name + ":");
            NodeList nodes = elementNode.getChildNodes();
            output(nodes);    //递归调用
```

```
            }
        }
    }
}
```

注意：和例 2 类似，本章后续的许多例子中将使用递归方法处理节点。

4.3 Document 节点

解析器的 parse()方法将整个被解析的 XML 文件封装成一个 Document 节点返回，应用程序可以从该节点的子孙节点中获取整个 XML 文件中数据的细节。

Document 节点可以有的两个直接子节点，其类型分别是 DocumentType 类型和 Element 类型，其中 DocumentType 节点对应着 XML 文件所关联的 DTD 文件，通过进一步获取该节点子孙节点来分析 DTD 文件中的数据；而其中的 Element 类型节点对应着 XML 文件的根节点，通过进一步获取该 Element 类型节点的子孙节点来分析 XML 文件中的数据。如果 XML 文件没有关联 DTD 文件，Document 节点就只有一个 Element 类型的子节点。

Document 节点经常使用下列方法获取和该节点相关的信息：

- Element getDocumentElement() 返回当前节点的 Element 子节点。
- DocumentType getDoctype() 返回当前节点的 DocumentType 子节点。
- NodeList getElementsByTagName(String name) 返回一个 NodeList 对象，该对象由当前节点的 Element 类型子孙节点组成，这些子孙节点的名字由参数 name 指定。
- NodeListgetElementsByTagNameNS(String namespaceURI, String localName) 返回一个 NodeList 对象，该对象由当前节点的 Element 类型子孙节点组成，这些子孙节点的名字由参数 localName 指定，名称空间由参数 namespaceURI 指定。
- String getXmlEncoding()返回 XML 文件使用的编码，即 XML 声明中 encoding 属性的值。
- boolean getXmlStandalone()返回 XML 声明中的 standalone 属性的值。
- String getXmlVersion()返回 XML 声明中的 version 属性的值。

与 Document 节点相关的例子可参见前面的例 1。

4.4 Element 节点

Element 节点是 Document 节点最重要的子孙节点，因为被解析的 XML 文件的标记对应着这样类型的节点。表示 Element 节点的常量是 Node. ELEMENT_NODE，一个节点调用

```
short getNodeType()
```

方法返回的值如果等于 Node. ELEMENT_NODE，那么该节点就是 Element 节点。

Element 节点经常使用下列方法获取和该节点相关的信息：

- String getTagName() 返回该节点的名称，该名称就是此节点对应的 XML 中的标记名称。
- String getAttribute(String name) 返回该节点中参数 name 指定的属性的值，该属性值是此节点对应的 XML 的标记中属性的值。
- NodeList getElementsByTagName(String name) 返回一个 NodeList 对象，该对象由当前节点的 Element 类型子孙节点组成，这些子孙节点的名字由参数 name 指定。
- NodeList getElementsByTagNameNS(String namespaceURI, String localName) 返回一个 NodeList 对象，该对象由当前节点的 Element 类型子孙节点组成，这些子孙节点的名字由参数 localName 指定，名称空间由参数 namespaceURI 指定。
- boolean hasAttribute(String name) 判断当前节点是否有名字是参数 name 指定的属性。
- boolean hasAttributeNS(String namespaceURI, String localName) 判断当前节点是否有名字是参数 name 指定、名称空间是 namespaceURI 指定的属性。

注意：getTagName()方法是 Element 接口中的方法，getNodeName()方法是 Element 接口从 Node 接口继承的方法。对于 Element 节点，getTagName()和 getNodeName()返回的都是 Element 节点对应的 XML 文件中标记的名称。

下面的例 3 中，JAXPThree.java 中的 Element 节点使用常用方法获得 XML 文件的有关数据。例 3 中的 Java 程序的运行效果如图 4.5 所示。

【例 3】

example4_3.xml

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<商品列表>
   <商品名称 分类 = "家电">
      电视机
   </商品名称>
   <商品名称 分类 = "服装">
      雅格尔西装
   </商品名称>
   <商品名称 分类 = "食品">
      东北大米
   </商品名称>
</商品列表>
```

图 4.5 Element 节点的常用方法

JAXPThree.java

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
public class JAXPThree{
   public static void main(String args[]){
      try { DocumentBuilderFactory factory =
```

```
                    DocumentBuilderFactory. newInstance();
                    DocumentBuilder domPaser = factory. newDocumentBuilder();
                        Document document = domPaser. parse(new File("example4_3. xml")) ;
                        Element root = document. getDocumentElement() ;
                        NodeList nodeList = root. getChildNodes();
                        int size = nodeList. getLength();
                        for(int k = 0;k < size;k++){
                            Node node = nodeList. item(k);
                            if(node. getNodeType() == Node. ELEMENT_NODE){
                                Element elementNode = (Element)node;
                                String name = elementNode. getNodeName();
                                String id = elementNode. getAttribute("分类");
                                String content = elementNode. getTextContent();
                                System. out. print(name);
                                System. out. print("(" + id + ")");
                                System. out. println(": " + content);
                            }
                        }
                }
                catch(Exception e){}
            }
        }
```

在上面的例3中,解析器获得的Document节点有一个Element子节点:root,该节点对应XML文件的根标记。

需要特别注意的是,root节点一共有7个子节点,其中有3个Element节点、4个Text节点。三个Element节点分别对应XML文件中根标记的三个名字是“商品名称”的子标记,四个Text节点的情况如下:

第一个Text节点对应XML文件中根标记的开始标签和第一个“商品名称”标记的开始标签之间形成的空白区,即

```
<商品列表>
```

与

```
<商品名称 分类="家电">
```

之间的空白区。

第二个Text节点对应XML文件中第一个“商品名称”标记的结束标签和第二个“商品名称”标记的开始标签之间形成的空白区,即

```
</商品名称>
```

与

```
<商品名称 分类="服装">
```

之间的空白区。

第三个Text节点对应XML文件中第二个“商品名称”标记的结束标签和第三个“商

品名称”标记的开始标签之间形成的空白区，即

```
</商品名称>
```

与

```
<商品名称 分类 = "食品">
```

之间的空白区。

第四个 Text 节点对应 XML 文件中第三个“商品名称”标记的结束标签和根标记结束标签之间形成的空白区，即

```
</商品名称>
```

与

```
</商品列表>
```

之间的空白区。

上述四个空白区都是为了使得 XML 文件看起来更美观而形成的，但解析器并不知道这一点，所以解析器仍然认为它们是有用的文本数据(由空格字符组成)。

另外三个 Element 节点各自都有一个 Text 节点(root 节点的孙节点)，分别对应着三个“商品标记”包含的文本数据：“电视机”、“雅格尔西装”和“东北大米”。

例 3 中的解析器获得的 Document 节点及其子孙节点如图 4.6 所示。

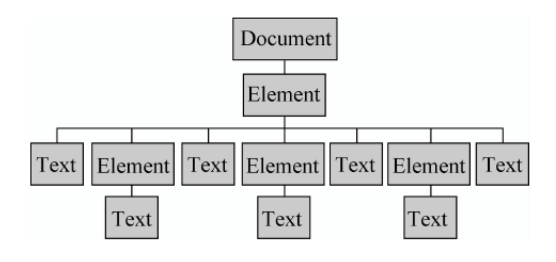


图 4.6 Document 节点的子孙节点

4.5 Text 节点

规范的 XML 文件的非空标记可以包含有子标记和文本内容。在 DOM 规范中，解析器使用 Element 节点封装标记，用 Text 节点封装标记的文本内容，即 Element 节点可以有 Element 子节点和 Text 节点。

表示 Text 节点的常量是 Node. TEXT_NODE，一个节点调用

```
short getNodeType()
```

方法返回的值如果等于 Node. TEXT_NODE，那么该节点就是 Text 节点。

Text 节点使用 String getWholeText()方法获取节点中的文本(包括其中的空白字符)。

注意：对于 Text 节点，getNodeName()方法返回的是“#text”。

让我们分析下列 name 标记：

```
<name> Jhon kelin
    <age> 35 </age>
    <sex> male</sex>
</name>
```

上述 name 标记将对应一个 Element 节点，而且所对应的这个 Element 节点将有 7 个子孙节点，其中 2 个 Element 子节点、3 个 Text 子节点和 2 个 Text 孙节点。这些子孙节点和 XML 中的标记及文本有如下的对应关系。

- 2 个 Element 子节点分别对应 name 标记的 2 个子标记：age 和 sex。
- 3 个 Text 子节点分别对应着："<name>"与"<age>"之间的文本、"Jhon kelin"与"<age>"之间的空白类字符、"</sex>"与"</name>"之间的空白类字符。
- 2 个 Text 孙节点分别对应 age 标记和 sex 标记包含的文本内容："35"和"male"。

下面的例 4 中，JAXPFour.java 中的 Text 节点使用常用方法获得文本数据。例 4 中的 Java 程序还计算了 Text 节点的数目，运行效果如图 4.7 所示。

【例 4】

example4_4.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<同学录>
   <姓名> 张大山
       <居住城市>大连市</居住城市>
       <联系电话> 0411 - 123456 </联系电话>
   </姓名>
   <姓名> 刘翠花
       <居住城市>北京市</居住城市>
       <联系电话> 010 - 654321 </联系电话>
   </姓名>
</同学录>
```

```
姓名: 张大山
  居住城市:大连市
  联系电话:0411-123456

姓名: 刘翠花
  居住城市:北京市
  联系电话:010-654321

一共有13个Text节点
```

图 4.7　Text 节点中的文本数据

JAXPFour.java

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
public class JAXPFour{
   public static void main(String args[]){
      GiveData give = new GiveData();
      try{ DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
            DocumentBuilder domPaser = factory.newDocumentBuilder();
            Document document = domPaser.parse(new File("example4_4.xml"));
            Element root = document.getDocumentElement();
            NodeList nodeList = root.getChildNodes();
            give.output(nodeList);
```

```
            System.out.println("一共有" + give.m + "个 Text 节点");
        }
        catch(Exception e){}
    }
}
class GiveData{
    int m = 0;
    public void output(NodeList nodeList){      //这是一个递归方法
        int size = nodeList.getLength();
        for(int k = 0;k < size;k++){
            Node node = nodeList.item(k);
            if(node.getNodeType() == Node.TEXT_NODE){
                Text textNode = (Text)node;
                String content = textNode.getWholeText();
                m++;
                System.out.print(content);
            }
            if(node.getNodeType() == Node.ELEMENT_NODE){
                Element elementNode = (Element)node;
                String name = elementNode.getNodeName();
                System.out.print(name + ":");
                NodeList nodes = elementNode.getChildNodes();
                output(nodes);
            }
        }
    }
}
```

对于应用程序而言，Text 节点是较重要的节点，因为 Text 节点封装着 XML 标记包含的文本数据。下面例 5 中的 XML 文件是关于电视机的价格信息，例 5 中的 Java 程序 JAXPFive.java 解析出这些数据，并利用所解析出的数据计算了电视机的平均价格，运行效果如图 4.8 所示。

【例 5】

example4_5.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<电视机列表>
    <名称> 海尔电视机
        <价格 单位 = '元/台'> 5238 </价格>
    </名称>
    <名称> 星海电视机
        <价格 单位 = '元/台'> 3660 </价格>
    </名称>
    <名称> 长虹电视机
        <价格 单位 = '元/台'> 5285 </价格>
    </名称>
</电视机列表>
```

```
  海尔电视机
    价格(元/台):5238
  星海电视机
    价格(元/台):3660
  长虹电视机
    价格(元/台):5285
平均价格:4727.67元/台
```

图 4.8 解析价格

JAXPFive.java

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
public class JAXPFive{
    public static void main(String args[]){
        GiveData give = new GiveData();
        try { DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();
                DocumentBuilder domPaser = factory.newDocumentBuilder();
                Document document = domPaser.parse(new File("example4_5.xml"));
                NodeList nodeList = document.getChildNodes();
                give.output(nodeList);
                System.out.printf("平均价格:%5.2f%s",give.average/give.m,give.mess);
        }
        catch(Exception e){
                System.out.println(e);
        }
    }
}
class GiveData{
      double average = 0,m = 0;
      String mess;
      public void output(NodeList nodeList){
           int size = nodeList.getLength();
           for(int k = 0;k < size;k++){
             Node node = nodeList.item(k);
             if(node.getNodeType() == Node.TEXT_NODE){
               Text textNode = (Text)node;
               String content = textNode.getWholeText();
               System.out.print(content);
               Element parent = (Element)textNode.getParentNode();
               boolean boo = (parent.getNodeName()).equals("价格");
               if(boo == true){
                    content = textNode.getWholeText();
                    average = average + Double.parseDouble(content.trim());
                    m++;
               }
             }
             if(node.getNodeType() == Node.ELEMENT_NODE){
               Element elementNode = (Element)node;
               String name = elementNode.getNodeName();
               String id = elementNode.getAttribute("单位");
               if(id.length()>0){
                    System.out.print(name + "(" + id + "): ");
                    mess = id;
               }
               NodeList nodes = elementNode.getChildNodes();
```

```
                output(nodes);
            }
        }
    }
}
```

4.6 Attr 节点

在 XML 文件中，属性并不是标记的子标记，因此，在 DOM 规范中，Attr 节点也不是 Element 节点的子节点。如果想解析 XML 文件中标记的属性，必须让对应的 Element 节点调用

```
NamedNodeMap getAttributes()
```

方法。该方法返回的 NamedNodeMap 对象由节点组成，这些节点可以被转换为 Attr 节点。Attr 节点通过调用 String getName()方法返回属性的名字，调用 String getValue()方法返回属性的值。

下面例 6 的 Java 程序输出了 Element 节点的属性及属性的值，运行效果如图 4.9 所示。

【例 6】

example4_6.xml

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<应聘者列表>
   <姓名 学历 = "研究生"> 张三
      <毕业学校 类型 = "工科">清华大学</毕业学校>
      <专业> 土木工程 </专业>
   </姓名>
   <姓名 学历 = "本科"> 张三
      <毕业学校  类型 = "理科">北京大学</毕业学校>
      <专业> 计算数学 </专业>
   </姓名>
</应聘者列表>
```

```
姓名:(学历:研究生) 张三
   毕业学校:(类型:工科)清华大学
   专业: 土木工程

姓名:(学历:本科) 张三
   毕业学校:(类型:理科)北京大学
   专业: 计算数学
```

图 4.9 获取属性的值

JAXPSix.java

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
public class JAXPSix{
    public static void main(String args[]){
        GiveData give = new GiveData();
        try { DocumentBuilderFactory factory =
              DocumentBuilderFactory.newInstance();
              DocumentBuilder domPaser = factory.newDocumentBuilder();
              Document document = domPaser.parse(new File("example4_6.xml")) ;
              Element root = document.getDocumentElement() ;
```

```
                NodeList nodeList = root.getChildNodes();
                give.output(nodeList);
            }
            catch(Exception e){
                System.out.println(e);
            }
        }
    }
    class GiveData{
        public void output(NodeList nodeList){
            int size = nodeList.getLength();
            for(int k = 0;k < size;k++){
              Node node = nodeList.item(k);
              if(node.getNodeType() == Node.TEXT_NODE){
                  Text textNode = (Text)node;
                  String content = textNode.getWholeText();
                  System.out.print(content);
              }
              if(node.getNodeType() == Node.ELEMENT_NODE){
                  Element elementNode = (Element)node;
                  String name = elementNode.getNodeName();
                  System.out.print(name + ":");
                  NamedNodeMap map = elementNode.getAttributes();
                  for(int m = 0;m < map.getLength();m++){
                      Attr attrNode = (Attr)map.item(m);
                      String attName = attrNode.getName();
                      String attValue = attrNode.getValue();
                      System.out.print("(" + attName + ":" + attValue + ")");
                  }
                  NodeList nodes = elementNode.getChildNodes();
                  output(nodes);
              }
            }
        }
    }
```

4.7 DocumentType 节点

解析器的 parse()方法将整个被解析的 XML 文件封装成一个 Document 节点返回，Document 节点的两个子节点的类型分别是 DocumentType 类型和 Element 类型，其中的 DocumentType 节点对应着 XML 文件所关联的 DTD 文件，通过进一步获取该节点的子孙节点来分析 DTD 文件中的数据。而其中的 Element 类型节点对应着 XML 文件的根节点，通过进一步获取该 Element 类型节点的子孙节点来分析 XML 文件中的数据。如果 XML 文件没有关联 DTD 文件，Document 节点就只有一个 Element 类型的子节点。Document 节点调用 getDoctype()返回当前节点的 DocumentType 子节点。

若 XML 文件中的 DOCTYPE 声明为：

```
<! DOCTYPE 北京站始发列车时刻表 PUBLIC " - //ISO985//China//ForXML/Ch" "time.dtd">
```

那么 DocumentType 节点调用 getName()方法返回的是：

```
北京站始发列车时刻表
```

调用 getPublicId()方法返回的是：

```
- //ISO985//China//ForXML/Ch
```

调用 getSystemId()方法返回的是：

```
time.dtd
```

DocumentType 节点调用 getInternalSubset()方法可以返回内部 DTD 的内容。

下面的例 7 通过 DocumentType 节点获取 DTD 的有关信息。例 7 中的 time.dtd 是 example4_7.xml 关联的外部 DTD 文件。例 7 中的 Java 程序输出了 DocumentType 节点中的数据，运行效果如图 4.10 所示。

【例 7】

time.dtd

```
<! ELEMENT 北京站始发列车时刻表 (客车 * ) >
<! ELEMENT 客车 (车次,开车时间)>
<! ELEMENT 车次 ( # PCDATA)>
<! ELEMENT 开车时间 ( # PCDATA)>
<! ATTLIST 客车 类别 CDATA # REQUIRED>
```

```
DTD名字：北京站始发列车时刻表
public标识：-//ISO985//China//ForXML/Ch
system标识：time.dtd
内部DTD：null
```

图 4.10 获取 DTD 的信息和内容

example4_7.xml

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<! DOCTYPE 北京站始发列车时刻表 PUBLIC " - //ISO985//China//ForXML/Ch" "time.dtd">
<北京站始发列车时刻表>
   <客车 类别 = "特快">
      <车次> T259 </车次>
      <开车时间> 18:38 </开车时间>
   </客车>
   <客车 类别 = "普快">
      <车次> K1257 </车次>
      <开车时间> 23:12 </开车时间>
   </客车>
</北京站始发列车时刻表>
```

JAXPSeven.java

```
import org.w3c.dom. * ;
import javax.xml.parsers. * ;
import java.io. * ;
public class JAXPSeven{
    public static void main(String args[ ]){
        try { DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
              DocumentBuilder domParser = factory.newDocumentBuilder();
```

```
                Document document = domParser.parse(new File("example4_7.xml")) ;
                DocumentType doctype = document.getDoctype();
                String DTDName = doctype.getName();
                System.out.println("DTD 名字: " + DTDName);
                String publicId = doctype.getPublicId();
                System.out.println("public 标识: " + publicId);
                String systemId = doctype.getSystemId();
                System.out.println("system 标识: " + systemId);
                String internalDTD  = doctype.getInternalSubset();
                System.out.println("内部 DTD: " + internalDTD);
            }
        catch(Exception e){}
    }
}
```

4.8 处理空白

不同标记的开始标签与结束标签之间的缩进区域是为了使得 XML 文件看起来更美观而形成的,但解析器并不知道这一点,所以解析器仍然认为它们是有用的文本数据(由空白类字符组成,如:\t\n\x0B\f\r)。

例如,对于下列简单的 XML 文件:

```
<?xml  version = "1.0"  encoding = "UTF-8" ?>
<root>
    问候
    <a> hello </a>
    <b>你好</b>
</root>
```

根节点对应的 Element 节点有 5 个 Text 子孙节点,其对应关系如下。

"<root>"与"<a>"之间的空白类字符和文本:"问候";

"<a>"与""之间的文本:"hello";

""与""之间的空白类字符;

""与""之间的文本:"你好";

""与"</root>"之间的空白类字符。

人们习惯上称标记之间的缩进区是可忽略空白,这实际上不是很准确,因为 XML 文件的标记可以包含有文本和子标记(或两者的混合内容),在这种情况下,标记之间的区域中就可能包含非空白的字符内容。

如果我们不允许标记混合内容,即标记只有子标记或文本,在这种情形下,称标记之间的缩进区域是可忽略空白就比较恰当,这些空白区确实是为了使得 XML 文件看起来更加美观而形成的,这也是它们存在的唯一目的。

如果想让 DOM 解析器忽略缩进空白,即这些缩进空白不在 Document 节点中形成 Text 子节点,那么 XML 文件必须是有效的,而且所关联的 DTD 文件必须约束 XML 文

件的标记不能有混合内容，同时 DocumentBuilderFactory 对象在给出 DOM 解析器之前，必须调用

```
setIgnoringElementContentWhitespace(boolean whitespace)
```

进行设置，将参数 whitespace 取值为 true（true 的作用是忽略空白）。

在下面的例 8 中，解析器将忽略缩进空白，这样，根标记对应的 Element 节点共有 4 个 Text 子孙节点，运行效果如图 4.11 所示。

【例 8】

bookList.dtd

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<! DOCTYPE 图书列表 SYSTEM "bookList.dtd">
<图书列表>
   <book>
      <书名>Java 程序设计</书名>
      <出版社>清华大学出版社</出版社>
   </book>
   <book>
      <书名>高等数学</书名>
      <出版社>高等教育出版社</出版社>
   </book>
</图书列表>
```

图 4.11 忽略缩进空白

example4_8.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<! DOCTYPE 图书列表 SYSTEM "bookList.dtd">
<图书列表>
   <book>
      <书名>Java 程序设计</书名>
      <出版社>清华大学出版社</出版社>
   </book>
   <book>
      <书名>高等数学</书名>
      <出版社>高等教育出版社</出版社>
   </book>
</图书列表>
```

JAXPEight.java

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
public class JAXPEight{
    public static void main(String args[]){
        GiveData give = new GiveData();
        try { DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
              factory.setIgnoringElementContentWhitespace(true); //忽略缩进空白
              DocumentBuilder domPaser =  factory. newDocumentBuilder();
```

```
                Document document = domPaser.parse(new File("example4_8.xml")) ;
                NodeList nodeList = document.getChildNodes();
                give.output(nodeList);
                System.out.printf("\n一共有%d个Text节点",give.m);
            }
            catch(Exception e){}
        }
    }
    class GiveData{
        int m = 0;
        public void output(NodeList nodeList){
            int size = nodeList.getLength();
            for(int k = 0;k<size;k++){
              Node node = nodeList.item(k);
              if(node.getNodeType() == Node.TEXT_NODE){
                  Text textNode = (Text)node;
                  String content = textNode.getWholeText();
                  m++;
                  System.out.println(content);
              }
              if(node.getNodeType() == Node.ELEMENT_NODE){
                  Element elementNode = (Element)node;
                  String name = elementNode.getNodeName();
                  NodeList nodes = elementNode.getChildNodes();
                  output(nodes);
              }
            }
        }
    }
```

4.9 验证规范性和有效性

JAXP 提供的解析器默认地检查 XML 文件是否是规范的,并不检查 XML 文件是否是有效的,也就是说,DOM 解析器调用 parse()方法时,如果 XML 文件是规范的,parse()方法就返回一个 Document 节点,否则将显示 XML 文件中不符合规范的错误信息。

即使 XML 文件关联了一个 DTD,解析器也并不检查 XML 文件是否是有效的,即不检查 XML 文件是否遵守该 DTD 规定的限制条件。如果想要检查一个 XML 文件是否有效,必须让 DocumentBuilderFactory 对象 factory 事先设置是否检查 XML 文件的有效性,如:

```
factory.setValidating(true);
```

有关的例题可参见第 3 章的 3.2 节。

4.10 使用 DOM 生成 XML 文件

对于一个给定的 XML 文件,通过使用解析器可以在内存中建立和 XML 结构相对应的树形结构数据。JAXP 也能让我们使用内存中的树形结构数据建立一个新的 XML 文件。

4.10.1 Transformer 对象

我们已经知道，解析器的 parse()方法将整个被解析的 XML 文件封装成一个 Document 节点返回，这就使得可以对 Document 节点进行修改，然后使用 Transformer 对象将一个 Document 节点变换为一个 XML 文件（Transformer 类在 javax.xml.transform 包中）。

解析器可不调用 parse()方法，而调用 newDocument()得到一个 Document 节点，例如：

```
Document document = domParse.newDocument();
```

应用程序可修改这样的 Document 节点，然后使用 Transformer 对象将这个 Document 节点变换为一个 XML 文件。

使用 Transformer 对象将一个 Document 节点变换为一个 XML 文件需要经过如下步骤。

(1) 使用 javax.xml.transform 包中的 TransformerFactory 类建立一个对象：

```
TransformerFactory transFactory = TransformerFactory.newInstance();
```

(2) 使用步骤(1) 中得到的 transFactory 对象调用 newTransformer()方法得到一个 Transformer 对象：

Transformer transformer=transFactory.newTransformer();

(3) 将被变换后得到的 Document 对象封装到一个 DOMSource 对象中(DOMSource 类在 javax.xml.transform.dom 包中)：

```
DOMSource domSource = new DOMSource(document);
```

(4) 将变换后得到的 XML 文件对象封装到一个 StreamResult 对象中(StreamResult 类在 javax.xml.transform.stream 包中)：

```
File file = new File("newXML.xml");
FileOutputStream out = new FileOutputStream(file);
StreamResult xmlResult = new StreamResult(out);
```

(5) Transformer 对象 transformer 调用 transform()方法实施变换：

```
transformer.transform(domSource,xmlResult);
```

4.10.2 用于修改 Document 的常用方法

Node 接口是 Document 接口的父接口，提供了许多用来修改、增加和删除节点的方法：

- Node appendChild(Node newChild) 节点调用该方法可以向当前节点增加一个新

的子节点,并返回这个新节点。

- Node removeChild(Node oldChild) throws DOMException 节点调用该方法删除参数指定的子节点,并返回被删除的子节点。
- Node replaceChild(Node newChild, Node oldChild) 节点调用该方法可以替换子节点,并返回被替换的子节点。

Element 接口本身除了从 Node 接口继承的方法外,还提供了用来增加节点的方法:

- Attr removeAttributeNode(Attr oldAttr) 删除 Element 节点的属性。
- void setAttribute(String name, String value) 为 Element 节点增加新的属性及属性值,如果该属性已经存在,新的属性将替换旧的属性。

Text 接口本身除了从 Node 接口继承的方法外,还提供了用来修改节点内容的方法:

- Text replaceWholeText(String content) 替换当前 Text 节点的文本内容。
- void appendData(String arg) 向当前 Text 节点尾加文本内容。
- void insertData(int offset, String arg) 向当前 Text 节点插入文本内容,插入的位置由参数 offset 指定,即第 offset 个字符的后继位置。
- void deleteData(int offset,int count) 删除当前节点的文本内容中的一部分,被删除的范围由参数 offset 和 count 指定,即从第 offset 个字符开始的后续 count 个字符。
- void replaceData(int offset,int count, String arg) 当前 Text 节点中文本内容的一部分替换为参数 arg 指定的内容,被替换的范围由参数 offset 和 count 指定,即从第 offset 个字符开始的后续 count 个字符。

4.10.3 用 DOM 建立 XML 文件

在下面的例 9 中,有一个关于银行营业时间的 example4_9.xml 文件,但是该 XML 文件中的“银行”标记没有名字为“关门时间”的子标记。例 9 中的 Java 程序解析 XML 文件 example4_9.xml,最后修改 Document 节点,为“银行”标记对应的 Element 节点增加新的对应“关门时间”标记的子节点,最后使用 Transformer 得到一个新的 XML 文件:newXML.xml。

【例 9】

example4_9.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<银行营业时间>
   <银行>
      <名称>建设银行</名称>
      <开门时间> 8:30 </开门时间>
   </银行>
   <银行>
      <名称>中国银行</名称>
      <开门时间> 9:30 </开门时间>
   </银行>
```

</银行营业时间>

JAXPNine.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.dom.*;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
public class JAXPNine{
    public static void main(String args[]){
        ModifyNode modify = new ModifyNode();
        try{ DocumentBuilderFactory factory =
              DocumentBuilderFactory.newInstance();
              DocumentBuilder domParser =  factory.newDocumentBuilder();
              Document document = domParser.parse(new File("example4_9.xml")) ;
              Element root = document.getDocumentElement() ;
              NodeList nodeList = root.getChildNodes();
              modify.modifyNode(nodeList,document);
              TransformerFactory transFactory = TransformerFactory.newInstance();
              Transformer transformer = transFactory.newTransformer();
              DOMSource domSource = new DOMSource(document);
              File file = new File("newXML.xml");
              FileOutputStream out = new FileOutputStream(file);
              StreamResult xmlResult = new StreamResult(out);
              transformer.transform(domSource,xmlResult);
              out.close();
        }
        catch(Exception e){
              System.out.println(e);
        }
      }
}
class ModifyNode{
    int m = 0;
    Document document;
    public void modifyNode(NodeList nodeList,Document document){
        this.document = document;
        int size = nodeList.getLength();
        for(int k = 0;k < size;k++){
          Node node = nodeList.item(k);
          if(node.getNodeType() == Node.ELEMENT_NODE){
              Element elementNode = (Element)node;
              String name = elementNode.getNodeName();
              if(name.equals("银行")){
                  Node textN = document.createTextNode("18:30");
                  Node elementN = document.createElement("关门时间");
```

```
                    elementN.appendChild(textN);
                    elementNode.appendChild(elementN);
                }
                NodeList nodes = elementNode.getChildNodes();
                modifyNode(nodes,document);
            }
        }
    }
}
```

上面例 9 中的 DOM 解析器利用已知的 XML 文件产生一个 Document 对象,然后对内存中的 Document 对象进行修改,生成如下的 XML 文件:newXML.xml。

newXML.xml

```
<?xml version = "1.0" encoding = "UTF - 8" standalone = "no"?>
<银行营业时间>
   <银行>
       <名称>建设银行</名称>
       <开门时间> 8:30 </开门时间>
       <关门时间> 18:30 </关门时间>
   </银行>
   <银行>
       <名称>中国银行</名称>
       <开门时间> 9:30 </开门时间>
       <关门时间> 18:30 </关门时间>
   </银行>
</银行营业时间>
```

在下面的例 10 中,DOM 解析器不再使用已经存在的 XML 文件,即不再去解析一个已存在的 XML 文件,而是调用 newDocument():

```
Document document = domPaser.newDocument();
```

得到一个 Document 节点,然后为 Document 节点增加新的子孙节点,最后使用 Transformer 得到一个新的 XML 文件 student.xml,用浏览器打开 student.xml 的效果如图 4.12 所示。

```
  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
- <学生列表>
  - <学生>
      <姓名>辛如学</姓名>
      <学号>201001</学号>
    </学生>
  - <学生>
      <姓名>纲尚学</姓名>
      <学号>201002</学号>
    </学生>
  - <学生>
      <姓名>楚进校</姓名>
      <学号>201003</学号>
    </学生>
  </学生列表>
```

图 4.12　用 DOM 生成的 XML 文件

【例 10】

JAXPTen.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.dom.*;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
public class JAXPTen{
    public static void main(String args[]){
        try { String [] studentName = {"辛如学","纲尚学","楚进校"};
              String [] studentNumber = {"201001","201002","201003"};
              DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
              DocumentBuilder domPaser = factory.newDocumentBuilder();
              Document document = domPaser.newDocument();
              document.setXmlVersion("1.0");
              Element root = document.createElement("学生列表");
              document.appendChild(root);
              for(int k = 1;k <= studentName.length;k++){
                  Node node = document.createElement("学生");
                  root.appendChild(node);
              }
              NodeList nodeList = document.getElementsByTagName("学生");
              int size = nodeList.getLength();
              for(int k = 0;k < size;k++){
                Node node = nodeList.item(k);
                if(node.getNodeType() == Node.ELEMENT_NODE){
                  Element elementNode = (Element)node;
                  Node nodeName = document.createElement("姓名");
                  Node nodeNumber = document.createElement("学号");
                  nodeName.appendChild(document.createTextNode(studentName[k]));
                  nodeNumber.appendChild(document.createTextNode(studentNumber[k]));
                  elementNode.appendChild(nodeName);
                  elementNode.appendChild(nodeNumber);
                }
             }
             TransformerFactory transFactory = TransformerFactory.newInstance();
             Transformer transformer = transFactory.newTransformer();
             DOMSource domSource = new DOMSource(document);
             File file = new File("student.xml");
             FileOutputStream out = new FileOutputStream(file);
             StreamResult xmlResult = new StreamResult(out);
             transformer.transform(domSource, xmlResult);
             out.close();
        }
        catch(Exception e){
             System.out.println(e);
        }
    }
}
```

习 题 4

1. Document 节点的两个子节点分别是什么类型?

2. 被解析的 XML 文件的标记和 Document 节点的哪种类型的子孙节点相对应?

3. 请编写一个 Java 程序(参考例 5),使用 DOM 解析器解析下列 XML 文件,要求输出各个标记的名字以及标记包含的文本数据,并计算出“数学”和“物理”的平均成绩。

Xiti3.xml

```
<?xml  version = "1.0" encoding = "UTF - 8" ?>
<成绩单>
   <张三>
         <数学> 89 </数学>
         <物理> 78 </物理>
   </张三>
   <李四>
         <数学> 67 </数学>
         <物理> 80 </物理>
   </李四>
</成绩单>
```

第5章 SAX解析器

主要内容

- 初识 SAX 解析器
- 文档开始与结束事件
- 标记开始与结束事件
- 文本事件
- 名称空间事件
- 错误事件
- 处理空白

上一章我们学习了基于 DOM 的解析器，通过 DOM 解析器应用程序，从 XML 文件中解析出所需要的数据。DOM 解析器的核心是在内存中建立和 XML 文件相对应的树形结构数据，XML 文件的标记、标记包含的文本内容等都会和内存中树形结构数据的某个节点相对应。使用 DOM 解析器的好处是一个应用程序可以方便地操作内存中树的节点来处理 XML 文档，获取所需要的数据。但 DOM 解析器也有不足之处，如果 XML 文件较大，相应的 Document 对象就要占用较多的内存空间，另外，应用程序可能并不需要 XML 文件的全部数据，而只是需要一部分，但为了获取这一部分数据却付出了较大的空间代价。

本章讲述基于事件的解析器：SAX 解析器。Sun 公司发布的 JDK1.5 的后续版本中提供了基于事件来解析 XML 文件的 API（支持 SAX 2.0）。和 DOM 解析器不同的是，SAX 解析器不在内存中建立和 XML 文件相对应的树形结构数据，其核心是事件处理机制。和 DOM 解析器相比，SAX 解析器占用的内存少，对于许多应用程序，使用 SAX 解析器来获取 XML 数据具有较高的效率。

DOM 解析器和 SAX 解析器各有所长和不足，不能说哪个解析器更好，只有很好地掌握了它们的工作原理，才能针对具体的问题选择恰当的解析器。

5.1 初识 SAX 解析器

5.1.1 SAX 解析器及其工作原理

SAX(Simple API for XML)提供了解析 XML 文件的 API，基于 SAX 的解析器称为

SAX解析器。SAX解析器的核心是事件处理机制,SAX解析器调用

```
parse(File f,DefaultHandler dh)
```

方法解析参数f指定的XML文件,并向该方法的参数dh传递一个事件处理器。SAX解析器在解析XML文件的过程中,根据从文件中解析出的数据产生相应的事件,并报告这个事件给事件处理器dh,事件处理器dh在处理事件时就会处理所发现的数据,parse()方法必须等待事件处理器dh处理事件完毕后再继续解析文件,报告下一个事件给事件处理器dh。当parse()方法在解析文件的过程中,发现已经解析完全部的内容,即发现了XML文件的根标记的结束标签时,将报告一个文档结束事件给处理器,不再继续解析文档,即结束parse()方法的执行。SAX解析器的工作原理如图5.1所示。

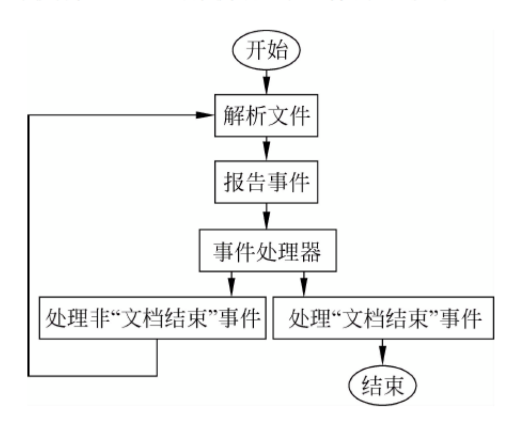


图5.1 SAX解析器的工作原理

从SAX解析器的工作原理可以看出SAX解析器较DOM解析器有更高的效率,这是因为事件处理器每次在内存中只保留对一个事件的处理,处理完毕即刻释放该处理过程所占用的内存。可以想象,若XML文件中许多标记包含的文本数据都含有相当多的字符,在这种情况下,使用DOM解析器将会在内存中建立占用较大内存的树形结构数据,导致过多地消耗系统的内存资源。如果使用SAX解析器,就可以轻松地获取这些标记包含的文本数据,因为SAX解析器把标记包含的文本数据按顺序分成若干个“文本”事件报告给事件处理器(见5.4节)。

5.1.2 创建SAX解析器的步骤与事件处理

首先简单地了解一下SAX解析器,有关细节将在后续小节分别讲述。

1. 创建SAX解析器的步骤

(1) 使用javax.xml.parsers包中的SAXParserFactory类调用其类方法newInstance()实例化一个SAXParserFactory对象:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

(2) 使用factory对象调用newSAXParser()方法返回一个SAXParser对象(SAXParser

类在 javax.xml.parsers 包中)：

```
SAXParser saxParser = factory.newSAXParser();
```

其中 SAXParser 对象称为 SAX 解析器。

2. 事件处理器

SAX 解析器使用下述 parse()方法解析 XML 文件：

- public void parse(File f,DefaultHandler dh) throws SAXException,IOException
- public void parse(InputStream is,DefaultHandler dh) throws SAXException,IOException
- public void parse(String uri,DefaultHandler dh) throws SAXException,IOException
- void parse(String uri) throws IOException,SAXException

上述 parse()方法的参数 dh 是 DefaultHandler 类型,解析器调用 parse()方法时,必须向 dh 传递一个 DefaultHandler 类或 DefaultHandler 类的子类的对象,DefaultHandler 类是 org.xml.sax.helpers 包中的类,该类或其子类的对象称为 SAX 解析器的事件处理器。SAX 解析器调用 parse()方法解析 XML 文件,根据从 XML 文件中解析出的数据产生相应的事件,并报告这个事件给事件处理器 dh,事件处理器 dh 就会处理所解析出的数据。DefaultHandler 类是实现了 ContentHandler、DTDHandler、EntityResolver 和 ErrorHandler 接口的类,也就是说,DefaultHandler 定义了依据 SAX 解析器报告的事件类型事件处理器 dh 应该调用的方法,例如,当解析器解析出一个标记的开始标签时,就将解析出的数据封装为一个"标记开始"事件,并报告该事件给事件处理器,事件处理器就会知道所发生的事件,然后对应地调用

```
startElement(String uri,String localName,String qName,Attributes atts)
```

方法对解析出的数据做出处理,方法中的参数 atts 是解析器发现的标记的全部属性,参数 uri 是解析器发现的标记的名称空间,localName 是标记的名称,qName 是带名称空间前缀的标记名称或标记的名称(依赖于该标记是否有名称空间)。如果标记没有名称空间,uri 值为 null(有关 startElement()方法的细节见 5.4 节)。

3. 事件的产生与处理

SAX 解析器的核心是事件处理机制,当 SAX 解析器调用 parse()方法解析 XML 文件时,事件处理器会根据所产生的事件类型调用相应的方法来处理发现的数据。在编写程序时,需要使用 DefaultHandler 类的子类创建一个事件处理器,当事件处理器对报告的事件类型不感兴趣时,就直接调用从父类继承的方法,采用默认的处理方式(父类方法的方法体中没有具体的处理语句),当事件处理器对报告的事件类型感兴趣时,子类就可以重写父类的某些方法,并调用重写的方法,以便事件处理器可以具体地处理解析器发现的数据。

在下面的例 1 中,通过解析一个简单的 XML 文件,观察事件处理器是如何处理解析器报告的有关事件的,有关细节将在后续的章节中讲述,例 1 中 SAXOne.java 的运行效果如图 5.2 所示。

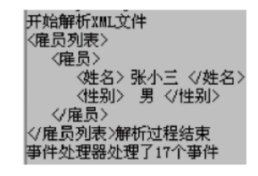


图 5.2　使用 SAX 解析器解析数据

【例 1】

example5_1. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<雇员列表>
   <雇员>
       <姓名> 张小三 </姓名>
       <性别> 男 </性别>
   </雇员>
</雇员列表>
```

SAXOne. java

```
import org.xml.sax.helpers. * ;
import org.xml.sax. * ;
import java.io. * ;
public class SAXOne{
   public static void main(String args[ ]){
       try{  File file = new File("example5_1.xml");
              SAXParserFactory factory = SAXParserFactory.newInstance() ;
              SAXParser saxParser = factory.newSAXParser();
              EventHandler handler = new EventHandler();  //事件处理器
              saxParser.parse(file,handler);
              System.out.println("事件处理器处理了" + handler.count + "个事件");
       }
       catch(Exception e){
              System.out.println(e);
       }
   }
}
class EventHandler extends DefaultHandler{
     int count = 0;
     public void startElement(String uri,String localName,String qName,Attributes atts){
        System.out.print("<" + qName + ">");
        count++;
     }
     public void endElement(String uri,String localName,String qName){
        System.out.print("</" + qName + ">");
        count++;
     }
     public void characters(char[ ] ch,int start,int length){
        String text = new String(ch,start,length);
        System.out.print(text);
        count++;
     }
     public void startDocument(){
        System.out.println("开始解析 XML 文件");
        count++;
     }
     public void endDocument(){
```

```
            System.out.println("解析过程结束");
            count++;
        }
    }
```

从上面的例1可以看出，解析器在调用parse()方法的过程中，事件处理器处理了17个如下的事件，如表5.1所示(按处理的顺序排列)。

表5.1 事件的产生与处理

序号	发现的数据	事件类型	事件处理器调用的方法
1	XML文件	文档开始	startDocument()
2	“雇员列表”开始标签：<雇员列表>	标记开始	startElement(String uri,String localName, String qName,Attributes atts)
3	<雇员列表>与<雇员>之间的空白字符	文本	characters(char[] ch,int start,int length)
4	“雇员”开始标签：<雇员>	标记开始	startElement(String uri,String localName, String qName,Attributes atts)
5	<雇员>与<姓名>之间的空白字符	文本	characters(char[] ch,int start,int length)
6	“姓名”开始标签：<姓名>	标记开始	startElement(String uri,String localName, String qName,Attributes atts)
7	“姓名”标记包含的文本内容	文本	characters(char[] ch,int start,int length)
8	“姓名”结束标签：</姓名>	标记结束	endElement(String uri,String localName,String qName)
9	</姓名>与<性别>之间的空白字符	文本	characters(char[] ch,int start,int length)
10	“性别”开始标签：<性别>	标记开始	startElement(String uri,String localName, String qName,Attributes atts)
11	“性别”标记包含的文本内容	文本	characters(char[] ch,int start,int length)
12	“性别”结束标签：</性别>	标记结束	endElement(String uri,String localName,String qName)
13	</性别>与</雇员>之间的空白字符	文本	characters(char[] ch,int start,int length)
14	“雇员”结束标签：</雇员>	标记结束	endElement(String uri,String localName,String qName)
15	</雇员>与</雇员列表>之间的空白字符	文本	characters(char[] ch,int start,int length)
16	“雇员列表”结束标签：</雇员列表>	标记结束	endElement(String uri,String localName,String qName)
17	XML文件结束	文档结束	endDocument()

注意：尽管相对DOM解析器，SAX解析器更加节省资源，但是，SAX解析器也有不足之处，例如，为了获取“性别”标记包含的文本内容，事件处理器要事先处理10个事件。

5.2 文档开始与结束事件

当解析器开始解析XML文件时，就会报告“文档开始”事件给事件处理器，然后再陆续地报告其他的事件，例如“标记开始”、“文本”事件等，最后报告“文档结束”事件。解析器报告“文档开始”事件，事件处理器就会调用startDocument()方法；解析器报告“文档结束”事件，事件处理器就会调用endDocument()方法。解析器在解析XML文件的过程中只能报告一次“文档开始”事件和“文档结束”事件。

下面的例2中,事件处理器重写父类的 startDocument()和 endDocument()方法,以便按照特殊的意图来处理“文档开始”事件和“文档结束”事件”。“文档开始”事件发生后,事件处理器显示 XML 文件的长度,“文档结束”事件发生后,事件处理器显示解析器解析整个 XML 文件所用的时间。在例2中,事件处理器对解析器报告的其他事件采用默认的处理方式,即事件处理器直接调用从父类继承的方法,这些方法的方法体里没有具体的语句,这样就减少了事件的处理时间。例2中 SAXTwo.java 的运行效果如图5.3所示。

【例2】

example5_2.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<图书信息>
   <图书>
      <名称>XML 基础教程(第 2 版)</名称>
      <价钱>26 元</价钱>
   </图书>
   <图书>
      <名称>JSP 基础教程(第 2 版)</名称>
      <价钱>28 元</价钱>
   </图书>
</图书信息>
```

开始解析XML文件
文件长度:297
解析过程结束
所用时间:0毫秒

图5.3 处理“文档”事件

SAXTwo.java

```
import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;
public class SAXTwo{
   public static void main(String args[]){
       try{  File file=new File("example5_2.xml");
             SAXParserFactory factory=SAXParserFactory.newInstance();
             SAXParser saxParser=factory.newSAXParser();
             EventHandler handler=new EventHandler(file);
             saxParser.parse(file,handler);
       }
       catch(Exception e){
             System.out.println(e);
       }
   }
}
class EventHandler extends DefaultHandler{
     File file;
     long timeStart=0,timeEnd=0;
     public EventHandler(File f){
        file=f;
```

```
        }
        public void startDocument(){
            timeStart = System.currentTimeMillis();
            System.out.println("开始解析 XML 文件");
            System.out.println("文件长度: " + file.length());
        }
        public void endDocument(){
            timeEnd = System.currentTimeMillis();
            System.out.println("解析过程结束");
            System.out.println("所用时间: " + (timeEnd - timeStart) + "毫秒");
        }
    }
```

5.3 标记开始与结束事件

当解析器发现一个标记的开始标签时，就将所发现的数据封装为一个“标记开始”事件，并报告该事件给事件处理器，事件处理器就会知道所发生的事件，然后调用

```
startElement(String uri,String localName,String qName,Attributes atts)
```

方法对发现的数据做出处理。参数 atts 是解析器发现的开始标签中定义的全部属性。

当 SAXParserFactory 对象设置为支持名称空间时，即相当于：

```
factory.setNamespaceAware(true);
```

上述方法中的参数 uri 的取值就是解析器发现的标记所隶属的名称空间的名字、参数 localName 的取值是标记的名称、参数 qName 的取值是带名称空间前缀的标记名称(如果有名称空间的前缀)或标记名称(如果没有名称空间的前缀)。

当 SAXParserFactory 对象未设置为支持名称空间时，即相当于：

```
factory.setNamespaceAware(false);
```

那么上述方法中的参数 uri 和 localName 的取值是空字符组成的串，即 uri="",localName="",参数 qName 的取值是标记名称。

事件处理器调用 startElement()方法后，将陆续地收到解析器报告其他的事件，例如“文本”事件，子标记的“标记开始”事件等。由于 XML 文件中的非空标记一定有结束标签，所以对同一个非空标记，解析器报告完该标记的“标记开始”事件后，一定还会报告该标记的“标记结束”事件，事件处理器就会知道所发生的事件，然后调用

```
endElement(String uri,String localName,String qName)
```

方法对发现的数据做出处理。

如果一个标记是空标记，解析器也报告“标记开始”事件和“标记结束”事件，即解析器将名字为“nullName”的空标记按不包含任何字符的非空标记：

```
<nullName></nullName>
```

来处理。

下面例 3 中的解析器解析出了 XML 文件中标记的个数以及标记的名称。例 3 中 SAXThree.java 的运行效果如图 5.4 所示。

【例 3】

example5_3.xml

```xml
<?xml version = "1.0" encoding = "UTF-8" ?>
<首都列表>
   <中国 xmlns:ch = "China">
      <ch:首都 区号 = "86"> 北京 </ch:首都>
   </中国>
   <美国 xmlns:US = "UnitedStates">
      <US:首都 区号 = "01"> 华盛顿 </US:首都>
   </美国>
   <南极 />
</首都列表>
```

```
<首都列表 >
<中国 >
<ch:首都 区号= "86">
标记隶属的名称空间是China
</ch:首都>
</中国>
<美国 >
<US:首都 区号= "01">
标记隶属的名称空间是UnitedStates
</US:首都>
</美国>
<南极 >
</南极>
</首都列表>

解析过程结束,共有6个标记
```

图 5.4 处理"标记"事件

SAXThree.java

```java
import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;
public class SAXThree{
   public static void main(String args[]){
      try { File file = new File("example5_3.xml");
            SAXParserFactory factory = SAXParserFactory.newInstance();
            factory.setNamespaceAware(true);
            SAXParser saxParser = factory.newSAXParser();
            EventHandler handler = new EventHandler();
            saxParser.parse(file,handler);
      }
      catch(Exception e){
            System.out.println(e);
      }
   }
}
class EventHandler extends DefaultHandler{
    int count = 0;
    public void startElement(String uri,String localName,
                             String qName,Attributes atts){
      count++;
      System.out.print("<" + qName + " ");
      for(int k = 0;k < atts.getLength();k++){
          System.out.print(atts.getLocalName(k) + " = ");
          System.out.print("\"" + atts.getValue(k) + "\"");
      }
```

```
        System.out.println(">");
        if(uri.length()>0)
            System.out.println("标记隶属的名称空间是" + uri);
    }
    public void endElement(String uri,String localName,String qName){
        System.out.println("</" + qName + ">");
    }
    public void endDocument(){
        System.out.printf("\n 解析过程结束,共有 %d 个标记",count);
    }
}
```

5.4 文本事件

XML 文件中非空标记包含的内容中可以有文本,当解析器解析这些文本数据时,就报告"文本"事件给事件处理器,事件处理器就会知道所发生的事件,然后调用

```
public void characters(char[] ch,int start,int length)
```

方法对标记包含的文本做出处理,参数字符数组 ch 中存放的就是该方法待处理的文本数据,start 是数组 ch 中存放字符的起始位置,length 是存放的字符的个数。需要注意的是,字符数组 ch 中存放的不一定是标记包含的全部文本,可能只是标记包含的全部文本的一部分。

标记的标签之间形成的缩进区域都是为了使得 XML 文件看起来更加美观而形成的,但解析器并不知道这一点,所以解析器仍然认为它们是有用的文本数据,当解析器发现这样的数据时,也会报告一个"文本"事件给事件处理器。

需要特别注意的是,对于标记包含的文本数据,解析器可能将这些文本分成几个连续的"文本"事件报告给事件处理器。例如,对于:

```
<hello>您好,很高兴认识您!</hello>
```

解析器将标记 hello 包含的文本数据,并用一个"文本"事件报告给事件处理器,"文本"事件报告的文本数据是:"您好,很高兴认识您!"。而对于

```
<hello>
    您好,
    很高兴认识您!
</hello>
```

解析器将标记 hello 中的文本数据分成 4 个"文本"事件报告给事件处理器。第一次"文本"事件报告的文本是"<hello>"和"您好,"之间的空白字符,第二次"文本"事件报告的文本是"您好,"以及后继的空白字符,第三次"文本"事件报告的文本是"很高兴认识您!",第四次"文本"事件报告的文本是"</hello>"之前的全部空白字符。

例 4 中统计了解析器报告的"文本"事件的次数以及所处理的文本,其中 SAXFour.java 的运行效果如图 5.5 所示。

【例 4】

example5_4.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<应聘者简历>
   <应聘者>
      <姓名>张三</姓名>
      <毕业学校>清华大学</毕业学校>
   </应聘者>
   <应聘者>
      <姓名>李四</姓名>
      <毕业学校>北京大学</毕业学校>
   </应聘者>
</应聘者简历>
```

```
第 1 次文本事件处理的文本是空白字符
第 2 次文本事件处理的文本是空白字符
第 3 次文本事件处理的文本是"张三"
第 4 次文本事件处理的文本是空白字符
第 5 次文本事件处理的文本是"清华大学"
第 6 次文本事件处理的文本是空白字符
第 7 次文本事件处理的文本是空白字符
第 8 次文本事件处理的文本是空白字符
第 9 次文本事件处理的文本是"李四"
第 10 次文本事件处理的文本是空白字符
第 11 次文本事件处理的文本是"北京大学"
第 12 次文本事件处理的文本是空白字符
第 13 次文本事件处理的文本是空白字符
```

图 5.5　处理“文本”事件

SAXFour.java

```
import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;
public class SAXFour{
   public static void main(String args[]){
      try { File file = new File("example5_4.xml");
            SAXParserFactory factory = SAXParserFactory.newInstance();
            SAXParser saxParser = factory.newSAXParser();
            EventHandler handler = new EventHandler();
            saxParser.parse(file,handler);
      }
      catch(Exception e){
            System.out.println(e);
      }
   }
}
class EventHandler extends DefaultHandler{
   int textEventCount;
   public void characters(char[ ] ch, int start, int length){
      textEventCount++;
      String text = new String(ch,start,length);
      text = text.trim();
      if(text.length() == 0)
        System.out.println("第 " + textEventCount + " 次文本事件处理的文本是空白字符");
      else
        System.out.println("第 " + textEventCount + " 次文本事件处理的文本是\"" + text + "\"");
   }
}
```

标记的文本数据是应用程序非常关心的数据,在下面的例 5 中,解析器分别输出了每个学生的总成绩,以及“数学成绩”和“英语成绩”的平均成绩。例 5 中 SAXFive.java 的运

行效果如图 5.6 所示。

【例 5】

example5_5. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<成绩单>
   <学生>
      <姓名> 张三</姓名>
      <数学成绩> 89 </数学成绩>
      <英语成绩> 77 </英语成绩>
   </学生>
   <学生>
      <姓名>李四</姓名>
      <数学成绩> 92 </数学成绩>
      <英语成绩> 65 </英语成绩>
   </学生>
</成绩单>
```

```
<成绩单>
   <学生>
      <姓名> 张三</姓名>
      <数学成绩> 89 </数学成绩>
      <英语成绩> 77 </英语成绩>
   </学生>该学生的总成绩：166.0
   <学生>
      <姓名>李四</姓名>
      <数学成绩> 92 </数学成绩>
      <英语成绩> 65 </英语成绩>
   </学生>该学生的总成绩：157.0
</成绩单>
共有2名学生
数学平均成绩：90.5
英语平均成绩：71.0
```

图 5.6 “文本”事件的有关计算

SAXFive. java

```
import javax.xml.parsers. * ;
import org.xml.sax.helpers. * ;
import org.xml.sax. * ;
import java.io. * ;
public class SAXFive{
   public static void main(String args[ ]){
      try { File file = new File("example5_5.xml");
            SAXParserFactory factory = SAXParserFactory.newInstance() ;
            SAXParser saxParser = factory.newSAXParser();
            EventHandler handler = new EventHandler();
            saxParser.parse(file,handler);
      }
      catch(Exception e){
            System.out.println(e);
      }
   }
}
class EventHandler extends DefaultHandler{
   boolean isComputabled,math,english;
   int count;
   double mathSum,englishSum,personSum;
   StringBuffer numberContent,otherContent;
   public void startElement(String uri,String localName,
                            String qName,Attributes atts){
      numberContent = new StringBuffer();
      otherContent = new StringBuffer();
      System.out.print("<" + qName + ">");
      if(qName.endsWith("成绩"))
         isComputabled = true;
```

```
        if(qName.startsWith("数学"))
          math = true;
        if(qName.startsWith("英语"))
          english = true;
        if(qName.equals("学生")){
            personSum = 0;
            count++;
        }
    }
    public void characters(char[ ] ch,int start,int length){
        String text = new String(ch,start,length);
        if(isComputabled == true)
            numberContent.append(text);
        System.out.print(text);
    }
    public void endElement(String uri,String localName,String qName){
        System.out.print("</" + qName + ">");
        if(isComputabled){
            String numberStr = new String(numberContent);
            numberStr = numberStr.trim();
            double d = Double.parseDouble(numberStr);
            personSum = personSum + d;
            if(math)
               mathSum = mathSum + d;
            if(english)
               englishSum = englishSum + d;
        }
        isComputabled = false;
        math = false;
        english = false;
        if(qName.equals("学生"))
          System.out.print("该学生的总成绩: " + personSum);
    }
    public void endDocument(){
         System.out.println("");
         System.out.println("共有" + count + "名学生");
         System.out.println("数学平均成绩: " + mathSum/count);
         System.out.println("英语平均成绩: " + englishSum/count);
    }
}
```

5.5 名称空间事件

我们曾在第 2 章详细讲解了名称空间,建立名称空间的目的是有效地区分名字相同的标记,当两个标记的名字相同时,它们可以通过隶属不同的名称空间来被相互区分。名称空间分为有前缀名称空间和无前缀名称空间,例如:

```
xmlns:tup = "http://www.tup.com"
```

声明的名称空间的名称是 http://www.tup.com，前缀是 tup。

```
xmlns = "http://www.cctv.com"
```

声明的名称空间的名称是 http://www.cctv.com，无前缀。

名称空间是在一个标记的开始标签中，当解析器在一个标记的开始标签中发现一个名称空间声明时，就先报告一个“名称空间开始”事件给事件处理器，然后再报告“标记开始”事件。解析器报告“名称空间开始”事件后，处理器就会知道所发生的事件，然后调用

```
public void startPrefixMapping(String prefix,String uri)
```

方法对发现的数据做出处理，方法中的参数 prefix 是解析器发现的名称空间的前缀，uri 是名称空间的名称，如果名称空间没有前缀，prefix 是不含任何字符的串，即 prefix=""。

名称空间涉及作用域的概念，一个标记如果使用了名称空间，那么该名称空间的作用域是该标记及其所有的子标记(见 2.9 节)。因此，解析器报告完“标记结束”事件后，就会报告一个“名称空间结束”事件给事件处理器，表明名称空间的作用域结束了，事件处理器就会知道所发生的事件，然后调用

```
public void endPrefixMapping(String prefix)
```

对发现的数据做出处理。例如，对于

```
<hello xmlns:tup = "www.tup.com">清华大学出版社</hello>
```

解析器报告事件的顺序是：“名称空间开始”事件、“标记开始”事件、“文本”事件、“标记结束”事件、“名称空间结束”事件。

为了能让解析器报告“名称空间开始”和“名称空间结束”事件，SAXParserFactory 需如下调用 setNamespaceAware(boolean b)方法：

```
factory.setNamespaceAware(true);
```

下面的例 6 中，example5_6 中的两个“图书”标记分别隶属名称为“清华大学出版社”和“机械工业出版社”的名称空间，SAXSix.java 输出了名称空间的名称和前缀，但只输出了隶属“清华大学出版社”的图书标记的子标记包含的文本数据。例 6 中 SAXSix.java 的运行效果如图 5.7 所示。

【例 6】

example5_6.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<图书列表 xmlns:清华 = "清华大学出版社"
          xmlns:机工 = "机械工业出版社">
    <清华:图书>
      <清华:书名>Java 程序设计</清华:书名>
      <清华:书名>JSP 程序设计</清华:书名>
   </清华:图书>
```

```
前缀:清华 名称空间的名称:清华大学出版社
前缀:机工 名称空间的名称:机械工业出版社
<清华:图书>
    <清华:书名>Java 程序设计</清华:书名>

    <清华:书名>JSP 程序设计</清华:书名>

  </清华:图书>

  解析过程结束,报告了2次名称空间
```

图 5.7 处理“名称空间”事件

```
  <机工:图书>
    <机工:书名>Java 程序设计</机工:书名>
    <机工:书名>JSP 程序设计</机工:书名>
  </机工:图书>
</图书列表>
```

SAXSix.java

```
import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;
public class SAXSix{
   public static void main(String args[]){
      try { File file = new File("example5_6.xml");
            SAXParserFactory factory = SAXParserFactory.newInstance();
            factory.setNamespaceAware(true);
            SAXParser saxParser = factory.newSAXParser();
            EventHandler handler = new EventHandler();
            saxParser.parse(file,handler);
      }
      catch(Exception e){
            System.out.println(e);
      }
   }
}
class EventHandler extends DefaultHandler{
   int count = 0;
   String uri;
   public void startPrefixMapping(String prefix,String uri){
       count++;
       System.out.print("前缀:" + prefix + " ");
       System.out.println("名称空间的名称:" + uri);
   }
   public void characters(char[] ch,int start,int length){
       String text = new String(ch,start,length);
       if(uri!=null&&uri.equals("清华大学出版社"))
           System.out.print(text);
   }
   public void startElement(String uri,String localName,
                             String qName,Attributes atts){
       this.uri = uri;
       if(uri.equals("清华大学出版社")){
           System.out.print("<" + qName + ">");
       }
   }
   public void endElement(String uri,String localName,String qName){
       if(uri.equals("清华大学出版社"))
         System.out.println("</" + qName + ">");
   }
```

```
    public void endDocument(){
        System.out.println("解析过程结束,报告了" + count + "次名称空间");
    }
}
```

5.6 错误事件

SAX 解析器默认地检查 XML 文件是否是规范的,若要使 SAX 解析器检查 XML 文件是否是有效的,SAXParserFactory 对象 factory 事先必须进行如下的设置:

```
factory.setValidating(true);
```

SAX 解析器在解析 XML 文件的过程中,如果发现错误就会报告一个"错误"事件给解析器,报告的信息是一个 SAXParseException 对象,事件处理器就会调用下列某个方法来处理错误信息:

- public void warning(SAXParseException e) throws SAXException
- public void error(SAXParseException e) throws SAXException
- public void fatalError(SAXParseException e) throws SAXException

上述三个方法中的参数 e 就是解析器报告的 SAXParseException 对象,这样,事件处理器在处理错误信息时,就可以让对象 e 调用相应的方法获取错误的细节,程序就可以输出错误的细节或将错误保存到文件中。

对象 e 调用

```
public String getMessage()
```

方法返回错误信息,调用

```
public void printStackTrace()
```

方法输出错误信息,调用

```
int getColumnNumber()
```

方法返回错误结尾所在的列号,调用

```
int getLineNumber()
```

方法返回错误结尾所在的行号,调用

```
public String getPublicId()
```

和

```
public String getSystemId()
```

方法可以获取有关的 PUBLIC 标识和 SYSTEM 标识。

上述三个方法 warning()、error()和 fatalError()都可以抛出 SAXException 异常,也就是说事件处理器在调用这些方法时,可以选择是否抛出一个 SAXException 对象给解析器。

解析器在调用 parse()方法时,必须使用 try…catch…语句来捕获 SAXException 异常,当 SAXException 异常发生时,parse()方法立刻结束执行,停止解析过程。例如,事件处理器在调用 warning()方法时,可以将有关的信息保存到文件中,但是,在保存时有可能发生 IOException 异常。这时,我们不希望解析器继续解析 XML 文件,那么,在处理 IOException 异常时,就可以抛出一个 SAXException 对象给解析器,解析器将停止 parse()方法的执行。

事件处理器在调用这些方法时,如果不抛出一个 SAXException 对象给解析器,解析器就会继续解析 XML 文件。如果解析器发现已无法继续解析文件,将停止 parse()方法的执行,并自动抛出一个 SAXException 异常给使用 SAX 解析器的应用程序。

当解析器报告一个“错误”事件后,事件处理器会根据错误的类型调用 warning()、error()和 fatalError()三个方法中的某一个,现在分别阐述如下。

1. warning(警告)

警告不属于规范性错误。当解析器认为有必要发出一个警告信息时,就报告一个“警告”事件给解析器,报告的信息是一个 SAXParseException 异常对象,事件处理器就会调用 warning()方法。处理器调用 warning()方法时,一般不必抛出 SAXException 异常,因为这些警告不会影响解析器继续解析文件。

2. error(一般错误)

一般错误属于非致命错误。解析器认为一般错误不会影响它继续解析文件,例如,当设置解析器检查 XML 文件的有效性时,解析器就会检查 XML 文件是否遵守了 DTD 文件所规定的约束,如果发现 XML 文件未遵守某个约束就会报告一个“一般错误”事件给解析器,报告的信息是一个 SAXParseException 异常对象。处理器调用 error()方法时,一般不必抛出 SAXException 异常,因为这些错误不会影响解析器继续解析文件。

3. fatalError(致命错误)

致命错误是最严重的错误。解析器认为这样的错误会影响它继续解析文件,因为这些错误导致继续解析 XML 文件完全是浪费时间,或解析无法进行。XML 文件不满足规范性或关联的 DTD 文件有错误等都会导致致命错误。所以,处理器调用 fatalError()方法时,应当抛出 SAXException 异常,停止解析过程。如果不抛出 SAXException 异常,让解析器继续解析文件,但解析器发现自己已经无法继续解析文件,那么它就会强制停止 parse()方法的执行。

下面的例 7 中,XML 文件不是有效的,它没有满足 DTD 的约束。例 7 中的 SAXSeven.java 输出了 XML 文件不是有效的有关信息,效果如图 5.8 所示。

【例 7】

sevenDTD.dtd

```
<!ELEMENT 商品信息 (商品*)>
<!ELEMENT 商品 (名称,生产日期,价格)>
<!ELEMENT 名称 (#PCDATA)>
<!ELEMENT 生产日期 (#PCDATA)>
<!ELEMENT 价格 (#PCDATA)>
```

```
开始解析XML文件
<商品信息>
   <商品>
      <名称> 电视机 </名称>
      <生产日期> 2010-12-20 </生产日期>
      <价格> 5676 元/台 </价格>
   </商品>
   <商品>
      <名称> 洗衣机 </名称>
      <价格> 6686 元/台 </价格>
      <生产日期> 2009-10-19 </生产日期>
   一般错误：The content of element type "商品" must match "(名称,生产日期,价格)
".位置：13,9
publicId:null
systemId:file:/C:/00xml/example5_7.xml
</商品>
</商品信息>解析过程结束
```

图 5.8 处理“错误”事件

example5_7.xml

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<! DOCTYPE 商品信息 SYSTEM  "sevenDTD.dtd">
<商品信息>
   <商品>
      <名称> 电视机 </名称>
      <生产日期> 2010-12-20 </生产日期>
      <价格> 5676 元/台 </价格>
   </商品>
   <商品> <! -- 下列三个标记的顺序不符合 DTD 约束(非致命错误) -->
      <名称> 洗衣机 </名称>
      <价格> 6686 元/台 </价格>
      <生产日期> 2009-10-19 </生产日期>
   </商品>
</商品信息>
```

SAXSeven.java

```
import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;
public class SAXSeven{
   public static void main(String args[]){
      try { File file = new File("example5_7.xml");
            SAXParserFactory factory = SAXParserFactory.newInstance();
            factory.setNamespaceAware(true);
            factory.setValidating(true);
            SAXParser saxParser = factory.newSAXParser();
            EventHandler handler = new EventHandler();
            saxParser.parse(file,handler);
      }
      catch(Exception e){
            System.out.println(e);
      }
   }
```

```
}
class EventHandler extends DefaultHandler{
    public void warning(SAXParseException e)throws SAXException{
        String warningMessage = e.getMessage();
        int row = e.getLineNumber();
        int colums = e.getColumnNumber();
        System.out.println("警告: " + warningMessage + "位置: " + row + "," + colums);
        System.out.println("publicId:" + e.getPublicId());
        System.out.println("systemId:" + e.getSystemId());
    }
    public void error(SAXParseException e) throws SAXException{
        String errorMessage = e.getMessage();
        int row = e.getLineNumber();
        int colums = e.getColumnNumber();
        System.out.println("一般错误: " + errorMessage + "位置: " + row + "," + colums);
        System.out.println("publicId:" + e.getPublicId());
        System.out.println("systemId:" + e.getSystemId());
    }
    public void fatalError(SAXParseException e) throws SAXException{
        String fatalErrorMessage = e.getMessage();
        int row = e.getLineNumber();
        int colums = e.getColumnNumber();
        System.out.println("致命错误: " + fatalErrorMessage + "位置: " + row + "," + colums);
        System.out.println("publicId:" + e.getPublicId());
        System.out.println("systemId:" + e.getSystemId());
        throw new SAXException("致命错误,停止解析");
    }
    public void startDocument(){
        System.out.println("开始解析 XML 文件");
    }
    public void endDocument(){
        System.out.println("解析过程结束");
    }
    public void startElement(String uri,String localName,
                              String qName,Attributes atts){
        System.out.print("<" + qName + ">");
    }
    public void endElement(String uri,String localName,String qName){
        System.out.print("</" + qName + ">");
    }
    public void characters(char[ ] ch,int start,int length){
        String text = new String(ch,start,length);
        System.out.print(text);
    }
    public void ignorableWhitespace(char[ ] ch,int start,int length){
        String text = new String(ch,start,length);
        System.out.print(text);
    }
}
```

5.7 处理空白

标记的标签之间的缩进区域都是为了使得 XML 文件看起来更美观而形成的，但解析器并不知道这一点，所以解析器仍然认为它们是有用的文本数据(由空白类字符组成)。

人们习惯上称标记的标签之间的缩进区是可忽略空白，这实际上不是很准确，因为 XML 文件的标记可以包含有文本和子标记(或两者的混合内容)，在这种情况下，标记的标签之间的区域中就可能包含可显示的字符。

如果我们不允许标记含有混合内容，即标记只包含有子标记或只包含有文本，在这种情形下，称标记的标签之间的缩进区域是可忽略空白就比较恰当，这些空白区使得 XML 文件看起来更加美观，是没有实际使用价值的空白字符。

SAX 解析器并不知道标记之间的缩进区域是为了使得 XML 文件看起来更美观，对于下面的 XML 文件：

```
<?xml  version  "1.0"  encoding = "UTF - 8"?>
<root>
    问候
    <a> hello</a>
    <b>你好</b>
</root>
```

解析器一共报告了 5 次“文本”事件给事件处理器。5 次“文本”事件报告的文本分别是：“<root>”与“<a>”之间的空白字符、“<a>”与“”之间的文本数据、“”与“”之间的空白字符、“”与“”之间的文本数据、“”与“</root>”之间的空白字符。

显然，我们不希望事件处理器去调用 characters()方法来处理标记的标签之间的缩进区域形成的这些空白字符，因为这会延迟事件处理器获取其他数据的时间。如果不想让事件处理器去调用 characters()方法来处理这些空白字符，那么 XML 文件必须是有效的，而且所关联的 DTD 文件必须规定 XML 文件的标记不能有混合内容。一旦这样做了，当解析器报告的“文本”事件属于标记的标签之间的缩进区域形成的空白类字符时，事件处理器就会去调用

```
ignorableWhitespace(char[ ] ch, int start, int length)
```

方法，而不去调用 characters()方法。

如果我们不准备处理这些空白，只要在编写 DefaultHandler 类的子类时，该子类直接从 DefaultHandler 类继承该方法即可(DefaultHandler 类的 ignorableWhitespace()方法不对空白字符做任何处理)。如果准备处理这些空白，只要在子类中重写 ignorableWhitespace()方法即可。

在下面的例 8 中 SAXEight.java 的解析器解析前面例 7 中的 example5_7.xml，在解析过程中使用 characters()方法处理标记包含的文本内容，使用 ignorableWhitespace()方法处理标记的标签之间的缩进区域形成的空白字符。例 8 中 SAXEight.java 的运行效

果如图 5.9 所示。

```
<商品信息>第1个空白区<商品>第2个空白区<名称> 电视机 </名称>第3个空白区<生产日期> 2010-12-20 </生产日
期>第4个空白区<价格> 5676 元/台 </价格>第5个空白区</商品>第6个空白区<商品>第7个空白区第8个空白区<名称
> 洗衣机 </名称>第9个空白区<价格> 6686 元/台 </价格>第10个空白区<生产日期> 2009-10-19 </生产日期>第11
个空白区</商品>第12个空白区</商品信息>解析过程结束,报告了12次空白
```

图 5.9　处理空白

【例 8】

SAXEight.java

```java
import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;
public class SAXEight{
   public static void main(String args[]){
       try {   File file = new File("example5_7.xml");
               SAXParserFactory factory = SAXParserFactory.newInstance() ;
               factory.setNamespaceAware(true);
               SAXParser saxParser = factory.newSAXParser();
               EventHandler handler = new  EventHandler();
               saxParser.parse(file,handler);
       }
       catch(Exception e){
               System.out.println(e);
       }
   }
}
class EventHandler extends DefaultHandler{
     int count = 0;
     public void characters(char[] ch,int start,int length){
          String text = new String(ch,start,length);
          System.out.print(text);
     }
     public void ignorableWhitespace(char[] ch,int start,int length){
          count++;
          System.out.print("第" + count + "个空白区");
     }
     public void startElement(String uri,String localName,
                              String qName,Attributes atts){
          System.out.print("<" + localName + ">");
     }
     public void endElement(String uri,String localName,String qName){
          System.out.print("</" + localName + ">");
     }
     public void endDocument(){
          System.out.println("解析过程结束,报告了" + count + "次空白");
     }
}
```

习 题 5

1. 对于下列 XML 文件,SAX 解析器报告事件的顺序是怎样的?

```
<?xml  version = "1.0"  encoding = "UTF - 8"  ?>
<商品信息>
      <商品>
          <名称>电视机</名称>
          <价钱> 3236 元</价钱>
      </商品>
</商品信息>
```

2. 如果解析器设置为忽略名称空间,

```
startElement(String uri,String localName,String qName, ,Attributes atts)
```

方法中各个参数的取值是怎样的?

3. 在什么情形下,解析器的事件处理器会调用

```
ignorableWhitespace(char[ ] ch, int start, int length)
```

方法?

4. 有如下的 XML 文件,请编写 Java 程序(参考例 5),使用 SAX 解析器获取标记中的文本数据,并根据这些数据计算出“货品列表”中全部货品的总重量。

Xiti4. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<货品列表>
    <货品>
        <名称> 电视</名称>
        <重量> 23.8 </重量>
    </货品>
    <货品>
        <姓名> 洗衣机</名称>
        <重量> 67.2 </重量>
    </货品>
</货品列表>
```

第 6 章 XPath 语言

主要内容

- XPath 简介
- Node 节点
- XPath 路径表达式的结构
- 谓词
- 节点集上使用谓词
- 节点集的并运算
- Java XPath API
- 节点集与函数
- 图书查询

XML 的核心是组织数据结构，有些文献甚至将 XML 称为 XML 数据库。自从 XML 诞生以来，XML 解析器的相关应用有着极其重要的地位，为此 W3C 为解析 XML 文件中的数据指定了 DOM 规范。

在第 4 章讲述了如何使用基于 DOM 规范的 DOM 解析器，其核心是在内存中创建和 XML 数据结构相对应的树形结构数据，不仅可以方便应用程序分析 XML 文件中的数据，而且应用程序也可以使用内存中的树形结构数据修改 XML 文件中的数据或创建新的 XML 文件。DOM 解析器的缺点是占用较多的内存，如果仅仅需要 XML 文件中的少量的特殊数据，使用 DOM 解析器就会事倍功半。

在第 5 章讲述了基于事件处理机制的 SAX 解析器，相对 DOM 解析器，SAX 解析器占用的内存少，对于许多应用程序，使用 SAX 解析器来获取 XML 数据具有较高的效率。但是 SAX 解析器也有不足之处，如果仅仅需要 XML 文件中的少量的特殊数据，使用 SAX 解析器可能需要事先处理一些不需要处理的事件后才可以获得应用程序想要的数据。

W3C 在 1999 年推出 XML Path Language (XPath) Version 1.0 规范，简称 XPath 1.0 语言规范，并在 2007 年对 XPath 1.0 进行了补充，正式公布了 XML Path Language (XPath) Version 2.0，简称 XPath 2.0 语言规范。XPath 1.0 是 XPath 的一个子集，XPath 2.0 中有大约 80％和 XPath 1.0 相同。

使用 XPath 可以很容易地编写查询 XML 中数据的 XPath 路径表达式，与 DOM 和 SAX 解析器的侧重点不同，XPath 语言为应用程序从 XML 文件中获得所需要的特殊数据提供了更加方便、快捷的语法，XPath 的作用非常类似于 SQL 语言在关系数据库中的作用。

本章将讲述 XPath 语言，读者也可以登录 http://www.w3.org/TR/xpathy.html 和 http://www.w3.org/TR/xpath20 了解完整的 XML Path Language (XPath) Version 1.0 和 XML Path Language (XPath) Version 2.0 规范。

由于 XML Path Language (XPath) Version 1.0 更加基础，而且目前 JDK1.6 中的 Java XPath API 是按照 XPath 1.0 实现的有关规范，因此本书侧重讲解 XML Path Language (XPath) Version 1.0。详细讲解 XML Path Language (XPath) Version 2.0 已经超出本书的范围，读者通过学习本书有助于学习 W3C 发布的 XPath 2.0 中对 XPath 1.0 所做的补充部分。

6.1 XPath 简介

XPath 语言的核心是给出用于从 XML 文件中查找标记的语法规则，即编写 XPath 路径表达式，以便使应用程序更加方便、快捷地从 XML 文件中检索到所需要的数据。本节简单介绍 XPath 路径表达式以及如何使用 XPath 路径表达式和 Java 提供的 API 编写检索 XML 中数据的应用程序，有关细节将在后续的小节中详细地讲述。

6.1.1 初识 XPath 路径表达式

一个 XPath 路径表达式，简称 XPath 表达式，由若干个“定位步”所构成（有关细节见 6.3 节）。XPath 路径表达式在表述形式上类似 UNIX 操作系统的文件系统路径的表述形式。

以下结合一个简单的 XML 文件来了解 XPath 路径表达式，语法的细节将在后续内容中讲解（见 6.3 节）。

【例 1】

example6_1.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<学生列表>
   <学生>
      <姓名> 张三 </姓名>
      <性别> 男 </性别>
      <成绩> 80 </成绩>
   </学生>
   <学生>
      <姓名> 李四 </姓名>
      <性别> 女 </性别>
      <成绩> 50 </成绩>
```

```
    </学生>
    <学生>
        <姓名> 孙伍 </姓名>
        <性别> 男 </性别>
        <成绩> 80 </成绩>
    </学生>
</学生列表>
```

下面是一个针对例 1 中的 XML 文件 example6_1.xml 的 XPath 路径表达式：

```
/学生列表/学生/姓名
```

该 XPath 路径表达式返回 example6_1.xml 中与该 XPath 路径表达式匹配的所有标记，即返回 example6_1.xml 文件中的所有名字为"姓名"的标记，下面的 XPath 路径表达式：

```
/学生列表/学生[2]/姓名
```

返回 example6_1.xml 文件中第 2 个名称为"学生"标记的所有"姓名"子标记，该 XPath 路径表达式的第 2 个定位步中使用了谓词(使用方括号给出的标记匹配条件)。

XPath 路径表达式的核心是给出一个匹配 XML 文件中标记的模式，也可以说 XPath 路径表达式的核心是表示满足一定条件的标记所组成的集合。

6.1.2 使用 XPath API

本节简单地介绍 Java 中处理 XPath 路径表达式的 API，细节将在后续内容中讲述(见 6.4 节)。使用 Java XPath API 处理 XPath 路径表达式的步骤如下。

(1) 使用 javax.xml.xpath 包中的 XPathFactory 类调用其类方法 newInstance()实例化一个 XPathFactory 对象：

```
XPathFactory xPathFactory = XPathFactory.newInstance();
```

(2) 将步骤(1)中得到的 XPathFactory 对象调用 newXPath()方法返回一个 XPath 对象：

```
XPath xPath = xPathFactory.newXPath();
```

(3) 使用 org.xml.sax 包中的 InputSource 类将 XML 文件封装到一个 InputSource 对象中：

```
InputSource source = new InputSource("student.xml");
```

(4) 将步骤(2)中获得的 XPath 对象调用 evaluate()方法来计算 XPath 路径表达式(evaluate()方法的细节见 6.3 节)：

```
NodeList nodelist =
(NodeList)xPath.evaluate("学生列表/学生[2]/姓名",source,XPathConstants.NODESET);
```

在下面的例 2 中,使用 Java XPath API 处理 XPath 路径表达式,输出 6.1.1 小节的 example6_1.xml 文件中成绩大于 60 的全部学生的姓名。例 2 中 XPathOne.java 的运行效果如图 6.1 所示。

【例 2】

XPathOne.java

```
import javax.xml.xpath.*;
import org.xml.sax.*;
import org.w3c.dom.*;
public class XPathOne{
   public static void main(String args[]){
     try{ XPathFactory xPathFactory = XPathFactory.newInstance();
          XPath xPath = xPathFactory.newXPath();
          InputSource source = new InputSource("example6_1.xml");
          String path = "/学生列表/学生[成绩>60]/姓名";
          NodeList nodelist =
        (NodeList)xPath.evaluate(path,source,XPathConstants.NODESET);
          int size = nodelist.getLength();
          for(int k = 0;k < size;k + + ){
               Node node = nodelist.item(k);
               String name = node.getNodeName();
               String content = node.getTextContent();
               System.out.print(name);
               System.out.println(":" + content);
          }
     }
     catch(Exception exp){
          System.out.println(exp);
     }
   }
}
```

姓名: 张三
姓名: 孙伍

图 6.1 使用 Java XPath API

6.2 Node 节点

和 DOM 规范类似,XPath 语言把 XML 文件中的标记、标记包含的文本等组成的数据结构看做是一个树形结构,即将 XML 文件看做是由 Node 类型节点构成的树。Node 节点又可细分为 Document、Element、Text、Attribute 等节点。本节之后,当提到节点时,如果没有特别指明其类型,就是指 Node 节点。

6.2.1 节点之间的关系

树形结构中的各个节点按其在树中的位置形成各种关系,例如一个节点是另一个节点的子节点等。和通常树形结构数据中使用的术语一样,XPath 语言也经常使用表明节点之间关系的术语:子节点、父节点、子孙节点、祖先节点、兄节点、弟节点、兄弟节点等。

一个节点的子节点也称为1级子节点，节点的1级子节点的子节点称为该节点的2级子节点……以此类推，节点的n级子节点的子节点称为该节点的n+1级子节点，将节点的任何级别的一个子节点也称为该节点一个子孙节点(descendant 节点)。

一个节点的父节点也称为1级父节点，节点的1级父节点的父节点称为该节点的2级父节点……以此类推，节点的n级父节点的父节点称为该节点的n+1级父节点，将节点的任何级别的一个父节点称为该节点一个祖先节点(ancestor 节点)。

一个节点的兄节点是指和该节点具有同样的级别，并且是该节点之前的某个节点；一个节点的弟节点是指和该节点具有同样的级别，并且是该节点之后的某个节点；一个节点的兄弟节点是指和该节点具有同样的级别的某个节点。

6.2.2 节点的类型

和DOM规范一样，XPath语言将XML文件看做是由Node类型节点构成的树，而且Node类型节点还可细分为Document、Element、Text、Attribute等节点。例如，XPath将下面的XML文件employee.xml看做是一个如图6.2所示的树形结构。

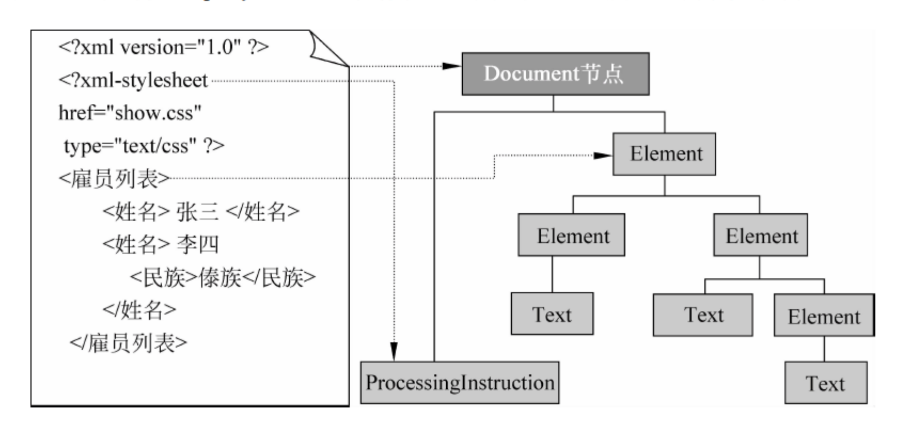


图6.2 employee.xml文件对应的Document节点

employee.xml

```
<?xml version = "1.0" ?>
<?xml - stylesheet href = "show.css" type = "text/css" ?>
<雇员列表>
   <姓名> 张三 </姓名>
   <姓名> 李四
       <民族>傣族</民族>
   </姓名>
</雇员列表>
```

1. Document(文档)节点

XPath语言将整个XML文件看做是一个树形结构的数据，并把XML文件作为该树的根节点，而且这个根节点的类型是Document类型的节点(上述图6.2中employee.xml

对应着的 Document 节点)。将 XML 文件中的处理指令、根标记、根标记的子孙标记以及标记包含的文本看做是 Document 根节点的子孙节点。因为 XML 文件只有一个根标记,因此 Document 节点有且仅有一个 Element 类型的节点,即这个 Element 节点对应着 XML 文件的根标记,XML 文件使用了几个操作指令,那么 Document 节点就会有对应的几个 ProcessingInstruction 子节点(注意,XML 声明不是操作指令)。

2. Element(元素)节点

XPath 语言将 XML 文件中的每个标记看做是一个 Element 节点。例如,对于上述 employee. xml,XPath 语言将 XML 文件中名称为"雇员列表"的标记看做是一个名字为"雇员列表"的 Element 节点(上述图 6.2 中"雇员列表"的标记对应着 Document 节点的 Element 子节点),而且该 Element 节点有两个名字都是"姓名"的 Element 子节点,分别对应着 employee. xml 文件中两个名字为"姓名"的标记。

规范的 XML 文件中的标记可以有子标记、文本以及所关联的属性,因此 Element 节点可以有 Element 子节点、Text 子节点。需要注意的是,在 XML 文件中属性并不是标记的子标记,因此,Attribute 节点也不是 Element 节点的子节点。

3. Text(文本)节点

XPath 语言将 XML 文件的标记中包含的文本看做是一个 Text 节点,而且该 Text 节点包含的文本内容就是它所对应的文本。Text 节点不能再有子节点(属于叶节点)。例如,对于上述 employee. xml,XPath 语言将"姓名"标记中的文本"张三"看做是一个 Text 节点。

4. Attribute(属性)节点

XPath 语言将 XML 文件中标记含有的属性看做是一个 Attribute 节点。由于 XML 文件中的标记和属性是关联关系,因此 Attribute 节点不是 Element 节点的子节点。但是需要注意的是,Atrribute 节点将与其关联的 Element 节点看做是自己的父节点,也就是说一个 Element 节点是它所关联的 Attribute 节点的父节点,但该 Attribute 节点却不是它的 Node 子节点。

5. ProcessingInstruction(操作指令)节点

XPath 语言将 XML 文件中的操作指令看做是一个 ProcessingInstruction 节点。ProcessingInstruction 节点名字就是操作指令的名字,其包含的文本内容就是操作内容的文本描述。因为 XML 文件的操作指令是写在 XML 文件的根标记的前面,所以 ProcessingInstruction 节点是 Document 节点的子节点,不是 Element 节点的子节点(如图 6.2 所示)。

6. Namespace 节点

XPath 语言将 XML 文件中,在标记的开始标签里声明的名称空间看做是一个 Namespace 节点,该节点的名字就是名称空间的前缀,节点包含的文本内容就是名称空间的名字。

因为 XML 文件不把名称空间看做是标记的子标记,因此 Namespace 节点不是 Element 节点的子节点。但需要注意的是,Namespace 节点将 Element 节点看做是自己的父节点,也就是说一个 Element 节点是 Namespace 节点的父节点,但该 Namespace 节

点却不是它的 Node 子节点。

7. Comment(注释)节点

XPath 语言将 XML 文件中的注释看做是一个 Comment 节点,注释节点包含的文本内容就是注释中的内容。Comment 节点可以是 Document 或 Element 的子节点。

6.2.3 节点的名字与值

Java APath API 使用对应的接口或类和 XPath 规范中的节点类型相对应(见第 4 章的 DOM 规范)。节点常用以下 3 个方法获取和它有关的信息。

- String getNodeName()获取节点的名字。
- String getNodeValue()获取节点的值。
- String getTextContent()获取和节点有关的文本内容。

上述 3 个方法返回的字符串依赖于调用方法的当前节点的类型,如表 6.1 所示。

表 6.1 节点的名字、值及包含的文本内容

节点类型	getNodeName()	getNodeValue()	getTextContent()
Document	"#document"	null	XML 文件中全部标记包含的文本内容
Element	标记的名字	null	节点以及它的子孙节点包含的文本内容
Text	"#text"	节点在 XML 中对应的文本	节点在 XML 中对应的文本
Attribute	属性的名字	属性的值	属性的值
ProcessingInstruction	操作指令的名字	操作指令中的操作内容	操作指令中的操作内容
Namespace	名称空间的前缀	名称空间	名称空间
Comment	"#comment"	注释的内容	注释的内容

需要注意的是,一个 Element 节点调用 getTextContent()返回当前节点以及它的子孙节点包含的文本内容。

6.3 XPath 路径表达式的结构

一个 XPath 路径表达式由若干个“定位步”构成,一个 XPath 路径表达式将返回一个节点集,即 XPath 路径表达式的核心是表示满足一定条件的标记所组成的集合。

6.3.1 绝对路径与相对路径

为了本节以及 6.3.2 小节和 6.3.3 小节讲解的方便,统一使用下面例 3 中的 XML 文件 example6_3.xml 阐述有关概念。

【例 3】

example6_3.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<studentList>
   <student xmlns:p1 = "Liaoning" sex = "男">
      <number>2010111
         <inputTime>1992 - 01 - 01</inputTime>
      </number>
       <name>张三</name>
      <birthDay>1992 - 01 - 01</birthDay>
      <score>611</score>
   </student>
   <student sex = "女">
      <number>2010222
         <inputTime>1992 - 01 - 01</inputTime>
      </number>
      <name>李翠花</name>
      <birthDay>1992 - 02 - 02</birthDay>
      <score>522</score>
   </student>
   <student sex = "男">
      <number>2010333
         <inputTime>1992 - 01 - 01</inputTime>
      </number>
      <name>孙五</name>
      <birthDay>1992 - 03 - 03</birthDay>
      <score>433</score>
   </student>
</studentList>
```

XPath 路径表达式由若干“定位步”从左向右用“/”连接而构成。XPath 路径表达式分为绝对路径和相对路径，从根节点开始(Document 节点)的路径表达式称为绝对路径，否则称为相对路径。

例如，对于上述例 3 中的 XML 文件 example6_3.xml，

```
/studentList/student/name
```

就是一个绝对路径，即用“/”开始的就是绝对路径。而

```
student/name
```

就是一个相对路径，即不用“/”开始的就是相对路径。

6.3.2 定位步与节点集

1. 定位步

“定位步”是构成 XPath 的基本单位，用于确定出相应的若干个节点，本节将详细讲

解各种"定位步"及其作用(读者请注意,以下用的标记名称均来自前面例 3 中的 example6_3.xml)。

一个定位步由轴(axis)、节点测试(node test)和可选的谓词所构成。

定位步的格式:

```
轴::节点测试 [谓词]
```

2. 定位步与节点的寻找

定位步的作用是找出节点。定位步中轴的作用是在树形结构数据中给出寻找 Node 节点的方向;节点测试的作用是给出寻找的 Node 节点应当属于哪种细分的类型,例如应当是 Element 或 Text 类型等;谓词的作用是给出所寻找的节点所需要满足的进一步的条件(定位步的谓词是可选项)。

在使用定位步的时候,必须给出一个该定位步的节点,称为该定位步的上下文节点(Context Node),简称当前节点。

当前节点使用定位步寻找节点,例如,对于定位步:

```
child::score
```

上述定位步的轴是 child,那么当前节点使用 child 轴寻找当前节点的全部 Node 类型子节点。节点测试 score 的作用是限定所寻找的子节点的名字必须是 score,类型必须是 Element 类型(当一个节点测试是标记的名字时,其作用是寻找 Element 类型节点,见 6.3.4 小节)。因此,当前节点使用定位步:

```
child::score
```

所寻找出的节点集就是当前节点的全部名字为 score 的 Element 类型子节点。

如果对寻找的节点有特殊的要求,就可以使用谓词,谓词和节点测试是所寻找的节点必须要满足的条件。例如,对于定位步:

```
child::score[position() = 2](缩写形式是 score[2])
```

使用上述定位步的当前节点首先要使用 child 轴和节点测试寻找出当前节点的名字是 score 的全部 Element 类型子节点,然后再使用谓词[position=2]从中筛选出第 2 个 Element 子节点(Element 节点的顺序就是指所对应的 XML 文件中标记出现的先后顺序)。

3. XPath 路径表达式与节点集

XPath 路径表达式由若干"定位步"从左向右用"/"连接而构成,XPath 绝对路径以根节点(Document 节点)开始。

XPath 绝对路径可以准确地确定出一个节点集。XPath 绝对路径是以 Document 节点作为第 1 个定位步,该定位步不需要有轴、谓词,习惯上称 Document 节点为虚节点,因为在 XPath 绝对路径中,无法显示地写出该节点。若绝对路径中的第 2 个定位步是 XML 文件的根标记的名字,那么 Document 节点使用该定位步确定出的节点集只含有一个 Element 节点;若绝对路径中的第 2 个定位步是 XML 文件的操作指令,那么 Document 节点使用该定位步确定出的节点集含有一个或几个 Processing-Instruction 节

点(如图 6.2 所示)。

XPath 绝对路径使用各个定位步最终寻找出一个节点集,寻找节点的规则如下:

假设 XPath 绝对路径经过第 n 个定位步后寻找出的第 n 个节点集中包含有 m 个节点,这 m 个节点按照它们在 XML 文件对应的顺序进行排列,序号从 1 开始,然后这 m 个节点依次使用第 n+1 个定位步寻找节点,它们找出的全部节点就是 XPath 绝对路径经过第 n+1 个定位步后寻找出的第 n+1 个节点集。以此类推,XPath 绝对路径经过最后一个定位步寻找出的节点集就是 XPath 绝对路径寻找出的节点集。

XPath 路径表达式寻找出的节点集也称为 XPath 路径表达式返回的节点集,或 XPath 路径表达式的"值"。

例如,对于 XPath 路径表达式:

```
/child::studentList/child::student/child::name(缩写/studentList/student/name)
```

(1) 第 1 个定位步

上述 XPath 绝对路径的第 1 个定位步是 Document 节点(即树的根节点,是虚节点)。

(2) 第 2 个定位步

上述 XPath 绝对路径的第 2 个定位步是:

```
child::studentList
```

由于 XML 文件只有一个名字为 studentList 的根标记,所以 Document 节点使用该定位步寻找出的节点集中有一个 Element 节点,该节点的名字是 studentList。

(3) 第 3 个定位步

上述 XPath 绝对路径的第 3 个定位步是:

```
child::student
```

XPath 路径在第 2 个定位步后寻找出的节点(只有 1 个 Element 节点,名字是 studentList)依次使用第 3 个定位步寻找节点,那么 XPath 路径经过第 3 个定位步寻找出的节点集中有 3 个 Element 节点,3 个 Element 节点的名字都是 student,3 个 Element 节点的排列顺序就是 XML 文件中标记名称为 student 的 3 个标记在 XML 文件中出现的先后顺序。

(4) 第 4 个定位步

上述 XPath 绝对路径的第 4 个定位步是:

```
child::name
```

XPath 路径在第 3 个定位步寻找出的节点(有 3 个 Element 节点,名字都是 student)依次使用第 4 个定位步寻找节点,那么 XPath 路径经过第 4 个定位步寻找出的节点集中有 3 个 Element 节点,3 个 Element 节点的名字都是 name,3 个 Element 节点的排列顺序就是 XML 文件中标记名称为 name 的 3 个标记在 XML 文件中出现的先后顺序。

简单地说,

```
/child::studentList/child::student/child::name
```

最终返回的节点集就是 XML 文件中名称是 name 的全部标记,但要求每个 name 标记的

1 级父标记的名字必须是 student,2 级父标记的名字必须是 studentList。

注意:我们需要将节点集中的节点看做是有顺序的,排列顺序是按照这些节点在 XML 文件中对应的操作指令、标记、文本或注释出现的先后顺序,因此,XPath 2.0 规范中将节点集称为一个序列,序列的每个分项是一个节点。

6.3.3 轴及缩写

XPath 路径表达式中的定位步使用轴来定位节点,而节点的具体类型由节点测试负责。轴是定位步的第一项,例如 child 轴用于寻找当前节点的全部子节点。

例如,当前节点使用定位步

```
child::student
```

寻找当前节点的全部名字为 student 的 Element 子节点。当前节点使用定位步:

```
child::student[position() = 2]
```

寻找当前节点的全部名字为 student 的 Element 子节点中的第 2 个 Element 子节点。

含有 child 轴的定位步:

```
chilld::节点测试
```

的缩写形式是:

```
节点测试
```

即省略“chilld::”。又如,

```
/child::studentList/child::student/child::name
```

的缩写是:

```
/studentList/student/name
```

表 6.2 是常用轴的缩写和简单的示例说明。

表 6.2 轴及缩写

轴名及用法	缩写	描述	示例
child::节点测试	节点测试	当前节点的 Node 子节点	/child::studentList/child::student/child::name 缩写: /studentList/student/name
descendant::节点测试	/节点测试	当前节点的 Node 子孙节点	/studentList/student/descendant::score 缩写: /studentList/student//score
parent::节点测试	..	当前节点的 Node 父节点	/studentList/student/name/parent::student 缩写: /studentList/student/name/..

续表

轴名及用法	缩写	描述	示　　例
ancestor::节点测试	没有缩写	当前节点的 Node 祖先节点	/studentList/student/name/ancestor::studentList//score
following::节点测试	没有缩写	当前节点的 Node 弟节点	/studentList/student/following::student/name
preceding::节点测试	没有缩写	当前节点的 Node 兄节点	/studentList/student/preceding::student/name
self::节点测试	.	当前 Node 节点	/studentList/student/name/self::name 缩写: /studentList/student/name/
attribute::属性名 attribute::*	@属性名 @*	当前节点所关联的 Attiribute 节点	/studentList/student/attribute::sex 缩写: /studentList/student/@sex
namespace::节点测试	没有缩写	当前节点所关联的 Namespace 节点	/studentList/student/namespace::p1

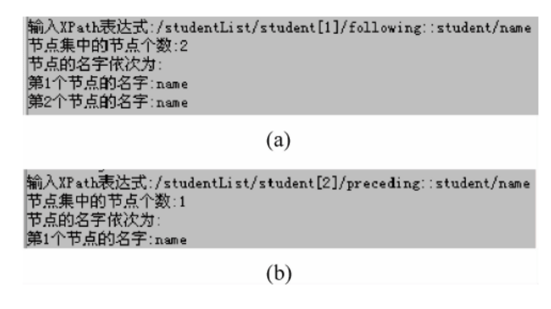
以下例 4 中的 Java 程序使用 Java XPath API 来测试表 6.2 中给出的轴，所使用 XML 文件是例 3 中的 example6_3.xml，有关 Java XPath API 细节将在 6.5 节讲述。在运行下列程序时，在命令行输入 XPath 路径表达式即可，图 6.3(a)是在命令行输入：

```
/studentList/student[1]/following::student/name
```

之后的运行效果。图 6.3(b)是在命令行输入：

```
/studentList/student[2]/following::student/name
```

之后的运行效果。



(a)

(b)

图 6.3　测试各种轴

【例 4】

XPathFour.java

```
import javax.xml.xpath.*;
import org.xml.sax.*;
import org.w3c.dom.*;
import java.io.*;
import java.util.*;
public class XPathFour{
   public static void main(String args[]){
     try{ XPathFactory xPathFactory = XPathFactory.newInstance();
          XPath xPath = xPathFactory.newXPath();
          Scanner reader = new Scanner(System.in);
          String fileName = "example6_3.xml";
          InputSource source = new InputSource(fileName);
          System.out.print("输入 XPath 表达式:");
          String path = reader.nextLine();
          NodeList nodelist =
          (NodeList)xPath.evaluate(path,source,XPathConstants.NODESET);
          int size = nodelist.getLength();
          System.out.println("节点集中的节点个数:" + size);
          System.out.println("节点的名字依次为:");
          for(int k = 0;k < size;k++){
              Node node = nodelist.item(k);
              String name = node.getNodeName();
              System.out.println("第" + (k + 1) + "个节点的名字:" + name);
          }
     }
     catch(Exception exp){
         System.out.println(exp);
     }
   }
}
```

6.3.4 节点测试

我们已经知道,XPath 路径表达式由若干"定位步"从左向右用"/"连接而构成路径,而定位步又是由轴(axis)、节点测试(node test)和可选的谓词所构成。使用定位步的当前节点根据轴确定所寻找的节点的方向,根据节点测试确定所寻找的节点的具体类型(有关节点的类型参见 6.2 节)。

一个节点测试可以是标记的名字、text()、node()、comment()等。例如,当一个节点测试是标记的名字时,其作用是寻找 Element 类型节点,即寻找 XML 文件中具有指定名称的标记。当节点测试是 text()时,其作用是寻找 Text 类型节点,即寻找 XML 文件中标记包含的文本。下面的表 6.3 给出了各种节点测试和作用。

表 6.3 节点测试和作用

节 点 测 试	作 用 描 述
标记的名字	寻找指定名字的 Element 类型节点
text()	寻找 Text 类型节点
node()	寻找 Node 类型节点
*	寻找任意名字的 Element 类型节点
processing-instruction()	寻找 Processing-Instruction 类型节点
comment()	寻找 Comment 类型节点

结合下面的例 5 中的 XML 文件 example6_5. xml 来讲解各种节点测试。

【例 5】

example6_5. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<列车时刻表>
   <列车 类别 = "特快" 车厢数目 = "20 节车厢">
      <列车号码> 152 次</列车号码>
      <始发时间> 09:12 </始发时间>
      <到达时间> 19:23 </到达时间>
      <始发站>北京</始发站>
      <终到站>上海</终到站>
   </列车>
   <列车 类别 = "普快" 车厢数目 = "12 节车厢">
      <列车号码> 168 次</列车号码>
      <始发时间> 10:12 </始发时间>
      <到达时间> 21:36 </到达时间>
      <始发站>沈阳</始发站>
      <终到站>南京</终到站>
   </列车>
</列车时刻表>
```

下面的例 6 中使用 Java XPath API 和例 5 中的 XML 文件来测试各种 XPath 路径表达式的效果，为了使用方便，例 6 采用了 GUI 界面。

【例 6】

XPathWindow. java

```
import javax.xml.xpath. * ;
import org.xml.sax. * ;
import org.w3c.dom. * ;
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
public class example6_5 {
   public static void main(String args[ ]){
      XPathWindow win = new XPathWindow("a.xml");
   }
}
```

```
class XPathWindow extends JFrame implements ActionListener{
   XPathFactory xPathFactory;
   XPath xPath;
   InputSource source;
   JTextField inputXPath;
   JTextArea showResult;
   JButton button;
   public XPathWindow(String fileName){
     try{ xPathFactory = XPathFactory.newInstance();
          xPath = xPathFactory.newXPath();
          source = new InputSource(fileName);
          inputXPath = new JTextField(25);
          showResult = new JTextArea();
          button = new JButton("确定");
          button.addActionListener(this);
          inputXPath.addActionListener(this);
          JPanel north = new JPanel();
          north.add(new Label("XPath 表达式:"));
          north.add(inputXPath);
          north.add(button);
          add(north,BorderLayout.NORTH);
          add(new JScrollPane(showResult),BorderLayout.CENTER);
          setBounds(10,10,900,300);
          setVisible(true);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     }
     catch(Exception exp){
         System.out.println(exp);
     }
   }
   public void actionPerformed(ActionEvent e){
      showResult.setText(null);
      String path = inputXPath.getText();
      try{
          NodeList nodelist =
          (NodeList)xPath.evaluate(path,source,XPathConstants.NODESET);
          int size = nodelist.getLength();
          showResult.append("节点集中的节点个数:" + size + "\n");
          showResult.append("节点的名字以及节点的值依次为:\n");
          for(int k = 0;k < size;k + + ){
              Node node = nodelist.item(k);
              String name = node.getNodeName();
              showResult.append("第" + (k + 1) + "个节点的名字:" + name + ",");
              String value = node.getNodeValue();
              showResult.append("第" + (k + 1) + "个节点的值:" + value + "\n");
          }
      }
      catch(Exception exp){
         showResult.setText(null);
```

```
            showResult.append("异常:" + exp);
        }
    }
}
```

以下是几种节点测试的 XPath 路径表达式的示例，读者可以运行上述例 6 中的 XPathWindow.java 来观察以下给出的 XPath 路径表达式中节点测试的效果。

(1) /child::node()(缩写：/node())

该路径表达式将返回 Document 节点的全部 Node 子节点(不包括 Atrribute 和 Namespace 节点，见 6.2 节阐述的节点关系)。在例 6 的 XPathWindow.java 运行界面的文本框中输入 XPath 路径表达式：/node()，然后单击“确定”按钮，程序的运行效果如图 6.4 所示。

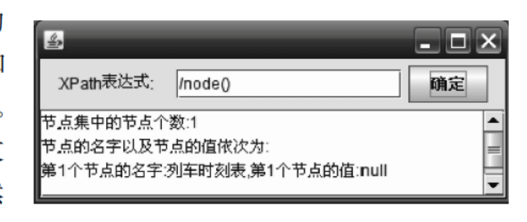


图 6.4 测试 XPath 路径表达式(一)

(2) /child::列车时刻表/child::列车(缩写：/列车时刻表/列车)

该路径表达式将返回“列车时刻表”节点的两个“列车”子节点。在例 6 的运行界面的文本框中输入 XPath 路径表达式：/列车时刻表/列车，然后单击“确定”按钮，程序的运行效果如图 6.5 所示。

(3) /child::列车时刻表/child::列车[position()=1]/child::*/child::text()(缩写：/列车时刻表/列车[1]/*/text())

该路径表达式将返回第一个“列车”节点的所有子孙 Text 节点。在例 6 的运行界面的文本框中输入 XPath 路径表达式：/列车时刻表/列车[1]/*/text()，然后单击“确定”按钮，程序的运行效果如图 6.6 所示。

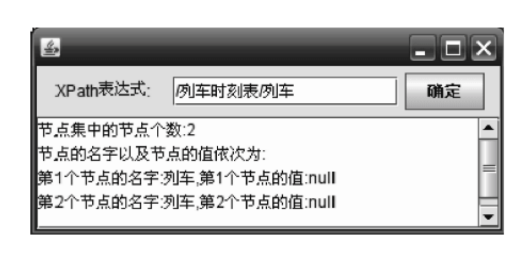


图 6.5 测试 XPath 路径表达式(二)

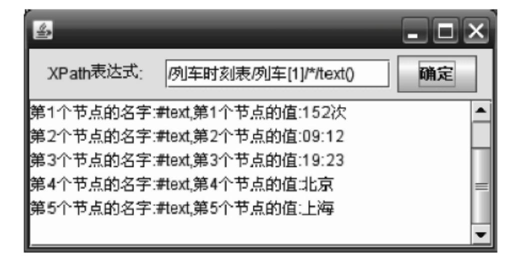


图 6.6 测试 XPath 路径表达式(三)

6.4 谓 词

谓词(Predicates)是定位步中最吸引人的部分，所以单独用一节来讲述。

6.4.1 谓词的格式与作用

谓词的目的是给出定位步所寻找出的节点需满足的进一步条件，即当前节点在使用

定位步的轴和节点测试找出若干个节点后,再使用谓词从这些节点中筛选出满足谓词条件的节点。谓词是用一对中括号括起来的条件表达式,谓词格式是:

```
[条件表达式]
```

谓词中的条件表达式是和节点有关的一个表达式,其值是 true 或 false,当条件表达式的值是 true 时,称节点满足谓词给出的条件,否则称节点不满足谓词给出的条件。

在谓词中使用 or,and 来表示逻辑关系,使用=,!=,<=,<,>=,>表示大小关系。例如:

```
[position()<= 4 and position()<= 8]
```

如果大小关系符比较的内容都是数字字符,关系运算将按数字比较大小,否则只能使用"="关系运算比较两个字符串是否相同。

我们已经知道 XPath 路径表达式由若干个"定位步"构成,而定位步的基本格式是:

```
轴::节点测试[谓词]
```

定位步的作用是让当前节点使用它寻找节点,一个 XPath 路径表达式每经过一个定位步都将寻找出一个节点集。

XPath 绝对路径的目的是寻找出一个节点集,其规则是:"XPath 绝对路径经过第 n 个定位步后寻找出的第 n 个节点集包含有 m 个节点,这 m 个节点按照它们在 XML 文件中对应的顺序进行排列,序号从 1 开始,然后这 m 个节点依次使用第 n+1 个定位步寻找节点,它们找出的全部节点就是 XPath 绝对路径经过第 n+1 个定位步后寻找出的第 n+1 个节点集。以此类推,XPath 绝对路径经过最后一个定位步寻找出的节点集合就是 XPath 绝对路径寻找出的节点集。"

使用带有谓词定位步:

```
轴::节点测试[谓词]
```

的当前节点首先根据轴和节点测试寻找出若干个节点,然后从这些节点中再筛选出满足谓词条件的节点,即筛选出使得谓词中条件表达式为 true 的节点。

例如,对于 XPath 路径表达式:

```
/child::列车时刻表/child::列车[position() = 2]/child::始发站/child::text()
(缩写: /列车时刻表/列车[2]/始发站/text())
```

(1) 第 1 个定位步

上述 XPath 绝对路径的第 1 个定位步是 Document 节点(即树的根节点,是虚节点)。

(2) 第 2 个定位步

```
child::列车时刻表
```

由于 XML 文件只有一个名字为"列车时刻表"根标记,所以 Document 节点使用该定位步寻找出的节点集中有一个 Element 节点,该节点的名字是"列车时刻表"。

(3) 第 3 个定位步

```
child::列车[2]
```

XPath 路径在第 2 个定位步寻找出的节点(只有一个 Element 节点)依次使用第 3 个定位步寻找 Element 节点,那么 XPath 路径经过第 3 个定位步寻找出的节点集中有 1 个 Element 节点(对应着 XML 文件中的第 2 个“列车”标记),这个 Elemnet 节点的名字是“列车”。

(4) 第 4 个定位步

```
child::始发站
```

XPath 路径在第 3 个定位步寻找出的节点(一个 Element 节点)依次使用第 4 个定位步寻找 Element 节点,那么 XPath 路径经过第 4 个定位步寻找出的节点集中有 1 个 Element 节点,这个 Elemnet 节点的名字是“始发站”(对应着 XML 文件中包含文本内容分别是“沈阳”的“始发站”标记)。

(5) 第 5 个定位步

```
child::text()
```

XPath 路径在第 4 个定位步寻找出的节点(1 个 Element 节点)依次使用第 5 个定位步寻找 Text 节点,那么 XPath 路径经过第 4 个定位步寻找出的节点集中有 1 个 Text 节点,这个 Text 节点的名字是“#text”,包含的文本内容是“沈阳”。

在例 6 的 XPathWindow.java 运行界面的文本框中输入 XPath 路径表达式:/列车时刻表/列车[2]/始发站/text(),然后单击“确定”按钮,程序的运行效果如图 6.7 所示。

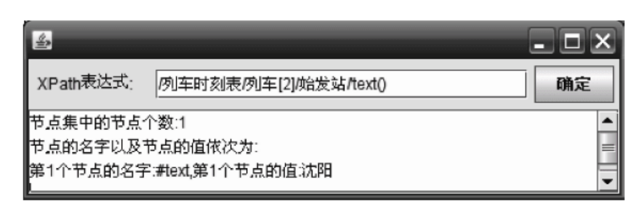


图 6.7 在定位步中使用谓词

6.4.2 寻找特殊位置的节点

在谓词中使用 position()和 last()函数可以寻找指定位置上的节点。

例如,对于定位步:

```
轴::节点测试[position() = 2](缩写: 轴::节点测试 谓词[2])
```

使用该定位步的当前节点所寻找的是满足轴和节点测试的全部节点中的第 2 个节点。

对于定位步:

```
轴::节点测试[last()]
```

使用该定位步的当前节点所寻找的是满足轴和节点测试的全部节点中的最后一个节点。

对于定位步:

```
轴::节点测试[last() - 1]
```

使用该定位步的当前节点所寻找的是满足轴和节点测试的全部节点中的倒数第 2 个节点。

对于定位步:

```
轴::节点测试[position()>=2 and position<=4]
```

使用该定位步的当前节点所寻找的是满足轴和节点测试的全部节点中位置大于等于 2 且小于等于 4 的节点。

对于定位步:

```
轴::节点测试[position() = 1 or position = 4]
```

使用该定位步的当前节点寻找满足轴和节点测试的全部节点中位置等于 1 或 4 的节点。

在前面例 6 的程序提供的用户输入文本框中输入 XPath 路径表达式:

```
/列车时刻表/列车[2]/*[position()>=1 and position()<=3]/text()
```

程序的运行效果如图 6.8 所示。

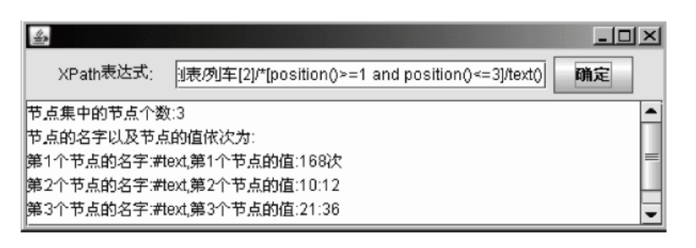


图 6.8 在定位步中使用位置谓词

6.4.3 寻找有特殊属性的节点

在谓词中使用 attribute 轴或@寻找具有指定属性或属性值的节点。

1. 寻找具有指定属性和属性值的节点

如果要寻找具有指定属性的节点,可以在谓词中使用:

```
attribute::属性名 1 逻辑关系符 … attribute::属性名 n
```

或使用缩写:

```
@属性名 1 逻辑关系符 … @属性名 n
```

例如,对于谓词:

```
[@类别 and @车厢数目]
```

那么满足该谓词条件的节点必须具有名字为“类别”和“车厢数目”属性的标记。

如果要寻找具有指定属性以及特定属性值的节点，可以在谓词中使用：

```
attribute::属性名1 大小关系符 '值' 逻辑关系符…attribute::属性名n 大小关系符 '值'
```

或使用缩写：

```
@属性名1 大小关系符 '值' 逻辑关系符…@属性名n 大小关系符 '值'
```

例如，下列 XPath 路径：

```
/列车时刻表/列车[@类别 and @车厢数目>=18]
```

寻找出的节点集中只有一个节点，该节点对应着 XML 文件中的“列车”标记，该标记具有“类别”属性和“车厢数目”属性，且“车厢数目”属性的值大于等于 18。

如果属性值是数字就可以用大小关系符对属性值进行判断比较，否则只能使用“＝”关系运算符比较属性值是否是某个字符串。例如：

```
[@类别='特快']
```

如果寻找具有属性，但对属性的名字没有特殊要求的节点，就可以使用谓词：

```
[attribute::*]
```

或使用缩写：

```
[@*]
```

任何具有属性的节点都满足上述谓词中的条件。

2. 使用 contains()函数

如果希望寻找具有指定属性，并且属性值中含有指定的字符串时，就可以在谓词中使用 contains()函数，格式是：

```
contains(@属性名,'特定字符串')
```

例如，对于定位步：

```
轴::节点测试[contains(attribute::属性名,'特定字符串')]
```

或

```
轴::节点测试[contains(@属性名,'特定字符串')]
```

使用该定位步的当前节点所寻找的是满足轴和节点测试的全部节点中具有指定属性名的属性，且属性值包含特定字符串的节点。

例如，对于定位步：

```
child::列车[contains(@类别,'快')]
```

使用该定位步的当前节点寻找当前节点的名字为“列车”的子节点，而且“列车”子节点必须具有属性“类别”、该属性值中含有字符“快”。

再例如,对于定位步:

```
child::列车[contains(@类别,'特') or contains(@车厢数目,'2')]
```

使用该定位步的当前节点寻找当前节点的名字为“列车”的子节点,而且“列车”子节点必须有属性“类别”、该属性值中含有字符“特”,或具有属性“车厢数目”、该属性值中含有字符“2”。

在例 6 的运行界面的文本框中输入 XPath 路径表达式:

```
/列车时刻表/child::列车[contains(@类别,'特')]
```

程序的运行效果如图 6.9 所示。



图 6.9 寻找具有指定属性的节点

6.4.4 寻找有特殊关系节点的节点

1. 寻找具有特殊关系节点的节点

如果要寻找具有特殊关系节点的节点,但对该特殊关系节点包含的内容没有特殊的要求,可以在谓词中使用:

```
轴::特殊关系节点名称 1 逻辑关系符 轴:: 特殊关系节点名称 2 … 轴:: 特殊关系节点名称 n
```

那么满足该谓词条件的节点必须和谓词中的节点形成轴所指定的特殊关系。例如,对于谓词:

```
[child::列车号码 and child::始发时间]
```

或缩写:

```
[列车号码 and 始发时间]
```

满足该谓词条件的节点必须有名字是“列车号码”和“始发时间”的子节点。

例如,对于谓词:

```
[following::到达时间]
```

满足该谓词条件的节点必须有名字是“到达时间”的弟节点。

如果要寻找具有特殊关系节点的节点,但对该特殊关系节点包含的内容有特殊的要求,可以在谓词中使用:

```
轴:: 特殊关系节点名称 1 大小关系 '值' 逻辑关系符 轴:: 特殊关系节点名称 n 大小关系 '值'
```

那么满足该谓词条件的节点必须和谓词中的节点形成轴所指定的特殊关系，且谓词中的节点所包含的文本必须满足谓词中给出的条件。例如，对于谓词：

```
[child::列车号码 = '152 次' and child::始发站 = '北京']
```

或缩写：

```
[列车号码 = '152 次' and 始发站 = '北京']
```

满足该谓词条件的节点必须有名字是“列车号码”和“始发站”的子节点，且“列车号码”子节点包含的文本必须是“152 次”，“始发站”子节点包含的文本必须是“北京”。

例如，对于谓词：

```
[following::始发站 = 'Beijing']
```

满足该谓词条件的节点必须有名字是“始发站”的弟节点，且该弟节点包含的文本必须是“Beijing”。

例如，对于谓词：

```
[child::text() = 'hello']
```

满足该谓词条件的节点必须有 Text 子节点，而且 Text 子节点包含的文本必须是“hello”。

如果节点包含的文本是数字就可以用大小关系符对文本内容进行判断比较，否则只能使用“＝”关系运算符比较文本是否是某个字符串。

在例 6 的运行界面的文本框中输入 XPath 路径表达式：

```
/列车时刻表/child::列车[child::列车号码 = '152 次' and child::始发站 = '北京']
```

或缩写：

```
/列车时刻表/列车[列车号码 = '152 次' and 始发站 = '北京']
```

程序的运行效果如图 6.10 所示。

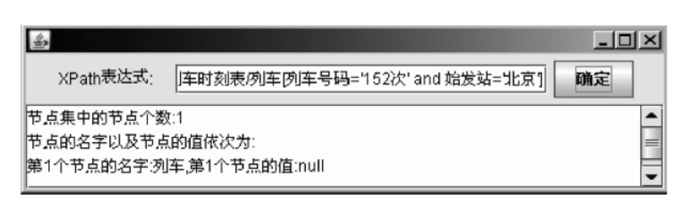


图 6.10 寻找具有指定特殊关系节点的节点

在例 6 的运行界面的文本框中输入 XPath 路径表达式：

```
/列车时刻表/ child::列车/ child::始发站[child::text() = '沈阳']/child::text()
```

或缩写：

```
/列车时刻表/列车/始发站[text() = '沈阳']/text()
```

程序的运行效果如图 6.11 所示。

图 6.11　寻找具有指定特殊关系节点的节点

2. 使用 contains()函数

如果要寻找具有特殊关系节点的节点,且特殊关系节点包含的文本需要含有指定的字符串时,就可以在谓词中使用 contains()函数,格式是:

```
contains(轴::节点名称,'特定字符串')
```

例如,对于谓词:

```
[contains(始发站,'京')]
```

满足该谓词条件的节点必须有名字为“始发站”的子节点,且该子节点包含的文本中含有“京”。

例如,对于谓词:

```
[contains(text(),'沈')]
```

满足该谓词条件的节点必须有 Text 子节点,且该 Text 节点包含的文本中含有“沈”。

在例 6 的运行界面的文本框中输入 XPath 路径表达式:

```
/列车时刻表/ child::列车/始发站[contains(child::text(),'沈')]/ child::text()
```

或缩写:

```
/列车时刻表/列车/始发站[contains(text(),'沈')]/text()
```

程序的运行效果如图 6.12 所示。

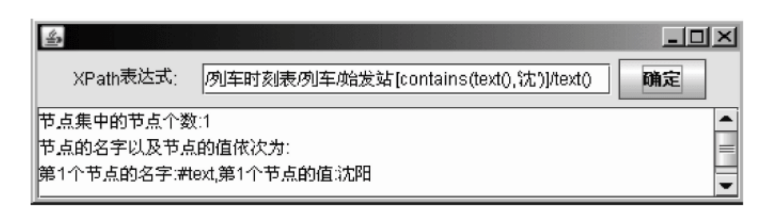


图 6.12　寻找具有指定特殊关系节点的节点

6.4.5　使用谓词嵌套寻找节点

谓词的作用是给出节点需满足的进一步条件,因此允许谓词中继续使用谓词。

例如，对于谓词：

```
[节点名[@属性名='特定值']]
```

满足上述谓词的节点需要有指定名字的子节点，而且子节点必须有指定的属性名和特定的属性值。

例如，对于谓词：

```
[following::节点名[@属性名='特定值']]
```

满足上述谓词的节点需要有指定名字的弟节点，而且弟节点必须有指定的属性名和属性值。

例如，对于谓词：

```
[descendant::节点名[@属性名='特定值']]
```

满足上述谓词的节点需要有指定名字的子孙节点，而且子孙节点必须有指定的属性名和特定的属性值。

在例 6 的运行界面的文本框中输入 XPath 路径表达式（节点名字是“列车”，且该“列车”节点具有名字是“始发时间”的子孙节点，且“始发站”节点包含文本“10：”）：

```
/列车时刻表/列车[descendant::始发时间[contains(child::text(),'10:')]]
```

程序的运行效果如图 6.13 所示。



图 6.13 谓词嵌套

注意：在使用谓词时，对于 descendant 轴不要使用缩写形式，因为缩写形式的“/”已不再是轴的名字或定位步的分割符，而是被看做一个运算符号，其级别却没有谓词“[]”的级别高，将导致节点顺序的混乱。

6.5 节点集上使用谓词

前面讲述了定位步中的谓词用法，XPath 允许在 XPath 绝对路径最后寻找出的节点集上使用谓词，其作用是从当前节点集中筛选、过滤出所需要的节点，即从当前节点集中筛选、过滤出一个子集。其使用格式是：

```
(绝对路径)[谓词]
```

例如：

```
(/列车时刻表/列车/始发站)[position()=1]
```

得到的节点集是 XPath 路径表达式：

/列车时刻表/列车/始发站

得到的节点集合中位置排序为 1 的节点构成的子集,即

(/列车时刻表/列车/始发站)[position() = 1]

得到的节点子集中只有一个节点,该节点名字为“始发站”,包含的文本内容是“北京”。

需要注意的是:

/列车时刻表/列车/始发站[position() = 1]

寻找节点的过程与

(/列车时刻表/列车/始发站)[position() = 1]

寻找节点的过程是不同的。

例如,按照绝对路径寻找节点的规则(见 6.3.2 小节),

/列车时刻表/列车/始发站[position() = 1]

寻找出的节点集合中有 2 个名字为“始发站”的节点(这 2 个“始发站”节点包含的文本内容分别是“北京”和“沈阳”)。

在例 6 的 XPathWindow.java 运行界面的文本框中分别输入 XPath 路径表达式:

/列车时刻表/列车/始发站[position() = 1]

和

(/列车时刻表/列车/始发站)[position() = 1]

观察程序运行结果的不同。

6.6 节点集的并运算

XPath 语言允许将多个 XPath 路径表达式寻找出的节点集合使用“|”进行集合的并运算,所得到的节点集中的节点按照节点在 XML 中的对应的标记或文本出现的先后顺序排列。例如,在例 6 中的 XPath 路径表达式输入框中输入

/列车时刻表/列车/始发时间/text()|/列车时刻表/列车/始发站/text()

程序的运行效果如图 6.14 所示。

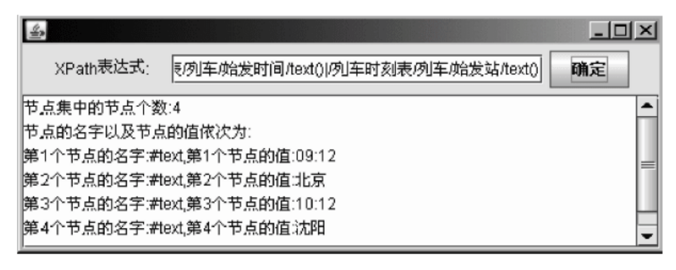


图 6.14 节点集的并运算

6.7 Java XPath API

在本章前面的 6.1.2 小节介绍了如何使用 Java XPath API 处理 XPath 路径表达式，其关键是使用 Java XPath API 提供的 XPath 对象调用 evaluate()方法来计算 XPath 路径表达式。

XPath 对象的 evaluate()方法的常用格式如下：

```
Object evaluate(String expression,InputSource source,QName returnType)
```

其中参数 expression 为 XPath 路径表式；参数 source 是指向 XML 文件的 InputSource 对象；参数 returnType 可以取值为：

```
XPathConstants.NODESET
XPathConstants.NODE
XPathConstants.STRING
XPathConstants.NUMBER
XPathConstants.BOOLEAN
```

以下根据参数 returnType 的 5 种取值情况分节讲述。

6.7.1 按 NodeSet 计算

当 XPath 对象调用

```
evaluate(String expression,InputSource source,QName returnType)
```

方法，并将参数 returnType 取值为：

```
XPathConstants.NODESET
```

时，evaluate()方法计算的结果是 org.w3c.dom 包中的一个 NodeList 对象，即 evaluate()方法返回 org.w3c.dom 包中的一个 NodeList 对象，该 NodeList 对象是由 org.w3c.dom 包中 Node 对象组成的一个节点集(有关 Node 和 NodeList 可参见第 4 章)，有关例题可参见前面的例 2、例 4 和例 6。

6.7.2 按 Node 计算

当 XPath 对象调用

```
evaluate(String expression,InputSource source,QName returnType)
```

方法，并将参数 returnType 取值为：

```
XPathConstants.NODE
```

时，evaluate()方法计算的结果是 org.w3c.dom 包中的一个 Node 对象，即 evaluate()方

法返回 org. w3c. dom 包中的一个 Node 对象，该 Node 对象对应着 XPath 路径表达式 expression 返回的节点集中的第一个节点。

下面的例 7 中使用了前面例 5 中的 XML 文件，XPath 对象在调用 evaluate()方法时，returnType 取值 XPathConstants. NODE，例 7 中的程序输出 XPath 路径表达式：

```
"/列车时刻表/列车"
```

得到的节点集中第一个节点的名字，以及该节点和全部子孙节点所包含的文本内容。例 7 中 XPathSeven. java 的运行效果如图 6. 15 所示。

【例 7】

```
列车:
    152次
    09:12
    19:23
    北京
    上海
```

图 6. 15　按 Node 计算节点集

XPathSeven. java

```
import javax.xml.xpath.*;
import org.xml.sax.*;
import org.w3c.dom.*;
public class XPathSeven{
   public static void main(String args[]){
      try{ XPathFactory xPathFactory = XPathFactory.newInstance();
            XPath xPath = xPathFactory.newXPath();
            InputSource source = new InputSource("example6_5.xml");
            String path = "/列车时刻表/列车";
            Node node = (Node)xPath.evaluate(path,source,XPathConstants.NODE);
            String name = node.getNodeName();
            String content = node.getTextContent();
            System.out.print(name);
            System.out.println(":" + content);
      }
      catch(Exception exp){
          System.out.println(exp);
      }
   }
}
```

注意：属性节点的名字就是属性的名称，属性节点包含的文本就是该属性的值。

6.7.3　按字符串计算

当 XPath 对象调用

```
evaluate(String expression,InputSource source,QName returnType)
```

方法，并将参数 returnType 取值为：

```
XPathConstants.String
```

时，evaluate()方法计算的结果是一个 String 对象，即 evaluate()方法返回一个 String 对象，该 String 对象对应着 XPath 路径表达式 expression 返回的节点集中的第一个节点包含的文本。

下面的例 8 中使用了前面例 5 中的 XML 文件，XPath 对象在调用 evaluate()方法时，returnType 取值 XPathConstants. String，例 8 中的程序输出 XPath 路径表达式：

```
"/列车时刻表/列车/始发站"
```

得到的节点集中第一个节点包含的文本内容，其运行效果如图 6.16 所示。

【例 8】

XPathEight. java

始发站标记包含的文本:
北京

图 6.16 按 STRING 计算节点集

```
import javax.xml.xpath.*;
import org.xml.sax.*;
import org.w3c.dom.*;
public class XPathEight{
   public static void main(String args[]){
     try{ XPathFactory xPathFactory = XPathFactory.newInstance();
          XPath xPath = xPathFactory.newXPath();
          InputSource source = new InputSource("example6_5.xml");
          String path = "/列车时刻表/列车/始发站";
          String stateName = (String)xPath.evaluate(path,source,XPathConstants.STRING);
          System.out.println("始发站标记包含的文本:");
          System.out.print(stateName);
     }
     catch(Exception exp){
          System.out.println(exp);
     }
   }
}
```

6.7.4 按布尔值计算

当 XPath 对象调用

```
evaluate(String expression,InputSource source,QName returnType)
```

方法，并将参数 returnType 取值为：

```
XPathConstants.BOOLEAN
```

时，evaluate()方法计算的结果是一个 Boolean 对象，即 evaluate()方法返回一个 Boolean 对象。如果 XPath 路径表达式 expression 返回的节点集为非空集合，evaluate 方法返回的 Boolean 对象中含有一个值为 true 的 boolean 类型数据，否则返回的 Boolean 对象中含有一个值为 false 的 boolean 类型数据。

6.7.5 按数值计算

当 XPath 对象调用

```
evaluate(String expression,InputSource source,QName returnType)
```

方法,并将参数 returnType 取值为:

```
XPathConstants.NUMBER
```

时,evaluate()方法计算的结果是一个 Double 对象,即 evaluate()方法返回一个 Double 对象,该 Double 对象对应着 XPath 路径表达式 expression 返回的节点集中的第一个节点包含的文本所转换成的 Double 对象,如果文本内容无法转化为 Double 对象,evaluate()方法返回"NaN",表示没有这样的数字。

下面的例 9 中有一个描述学生成绩的 XML 文件:example6_9.xml,例 9 中的程序在使用 XPath 对象调用 evaluate()方法时,returnType 取值 XPathConstants.NUMBER,程序输出学生的平均成绩,其运行效果如图 6.17 所示。

【例 9】

example6_9.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<成绩表>
   <学生>
       <姓名>张三</姓名>
       <成绩> 78 </成绩>
   </学生>
   <学生>
       <姓名>李四</姓名>
       <成绩> 80 </成绩>
   </学生>
</成绩表>
```

学生的平均成绩79.0

图 6.17 按 NUMBER 计算节点集

XPathNine.java

```
import javax.xml.xpath.*;
import javax.xml.xpath.*;
import org.xml.sax.*;
import org.w3c.dom.*;
public class XPathNine{
  public static void main(String args[]){
      try{ XPathFactory xPathFactory = XPathFactory.newInstance();
          XPath xPath = xPathFactory.newXPath();
          InputSource source = new InputSource("example6_9.xml");
          double sum = 0;
           String path = "成绩表/学生";
           NodeList nodelist =
          (NodeList)xPath.evaluate(path,source,XPathConstants.NODESET);
          int size = nodelist.getLength();
          for(int i = 1;i <= size;i++){
            path = "成绩表/学生[" + i + "]/成绩";
            Double number = (Double)xPath.evaluate(path,source,XPathConstants.NUMBER);
            double score = number.doubleValue();
```

```
            sum = sum + score;
          }
          double aver = sum/size;
          System.out.println("学生的平均成绩" + aver);
      }
      catch(Exception exp){
          System.out.println(exp);
      }
   }
}
```

6.8　节点集与函数

为了计算和节点集有关的数据，XPath 语言给出了几个作用于节点集上的函数。

1. count(node-set)函数

count(node-set)函数返回参数指定的节点集中的节点的个数，例如，

```
count(/列车时刻表/列车/始发时间)
```

返回的值是 2。

2. sum(node-set)函数

sum()函数将节点集中的节点包含的文本转换为数字，并返回它们的和。如果节点集中的某个节点包含的文本无法转换为数字，sum()返回“NaN”。

下面例 10 中的 XPathTen.java 在 XPath 路径表达式返回的节点集上使用 sum()函数计算了例 9 中 example6_9.xml 文件中全部学生的成绩之和，以及平均成绩。

XPathTen.java 运行效果如图 6.18 所示。

【例 10】

总成绩:158.0
平均成绩:79.0

图 6.18　使用 sum()函数

XPathTen.java

```
import javax.xml.xpath.*;
import org.xml.sax.*;
import org.w3c.dom.*;
public class XPathTen{
    public static void main(String args[]){
      try{ XPathFactory xPathFactory = XPathFactory.newInstance();
           XPath xPath = xPathFactory.newXPath();
           InputSource source = new InputSource("example6_9.xml");
           String countPath = "count(/成绩表/学生/成绩)";
           Double number = (Double) xPath.evaluate(countPath, source, XPathConstants.
NUMBER);
           double n = number.doubleValue();
           String sumPath = "sum(/成绩表/学生/成绩)";
           Double sum = (Double)xPath.evaluate(sumPath,source,XPathConstants.NUMBER);
           double total = sum.doubleValue();
           System.out.println("总成绩:" + total);
```

```
            double aver = total/n;
            System.out.println("平均成绩:" + aver);
        }
        catch(Exception exp){
            System.out.println(exp);
        }
    }
}
```

6.9 图书查询

本节使用 XPath 路径表达式查询一个描述图书信息的 XML 文件,在下面的例 11 中:XML 文件 book.xml 描述图书信息;BookConditionWindow.java 负责提供输入查询条件的界面;FindBookByXPath.java 负责获取 BookConditionWindow.java 界面中输入的查询条件,并根据该条件给出 XPath 路径表达式查询 book.xml;Application.java 是含有 main()方法的 Java 应用程序。运行 Application.java 的效果如图 6.19 所示。



图 6.19 图书查询

【例 11】

book.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<图书列表>
   <图书 ISBN = '72349876'>
      <名称>美丽的假日</名称>
      <作者> 张三 </作者>
      <价格> 29    </价格>
      <出版时间> 2009.05 </出版时间>
      <出版社> 阳光出版社</出版社>
   </图书>
   <图书 ISBN = '12345678'>
      <名称>冬天的阳光</名称>
      <作者> 李四 </作者>
      <价格> 18   </价格>
      <出版时间> 2010.05 </出版时间>
      <出版社> 冬冬出版社</出版社>
   </图书>
   <图书 ISBN = '87654321'>
      <名称>春天的记忆</名称>
```

```
        <作者> 张小民 </作者>
        <价格> 25  </价格>
        <出版时间> 2010.10 </出版时间>
        <出版社> 春日出版社</出版社>
    </图书>
</图书列表>
```

Application.java

```java
public class Application{
    public static void main(String args[]){
        new BookConditionWindow();
    }
}
```

BookConditionWindow.java

```java
import java.awt.*;
import javax.swing.*;
public class BookConditionWindow extends JFrame {
    JTextField inputBookName,inputBookAuthor,inputBookISBN,inputPublish;
    JTextArea showResult;
    JButton button;
    Box baseBox ,boxV1,boxV2;
    BookConditionWindow(){
        inputBookName = new JTextField(10);
        inputBookAuthor = new JTextField(10);
        inputBookISBN = new JTextField(10);
        inputPublish = new JTextField(10);
        boxV1 = Box.createVerticalBox();
        boxV1.add(new Label("图书名称中包含:"));
        boxV1.add(new Label("作者姓名中包含:"));
        boxV1.add(new Label("图书 ISBN 中包含"));
        boxV1.add(new Label("出版社名称中包含:"));
        boxV2 = Box.createVerticalBox();
        boxV2.add(inputBookName);
        boxV2.add(inputBookAuthor);
        boxV2.add(inputBookISBN);
        boxV2.add(inputPublish);
        baseBox = Box.createHorizontalBox();
        baseBox.add(boxV1);
        baseBox.add(boxV2);
        JPanel west = new JPanel();
        west.add(baseBox);
        button = new JButton("确定");
        west.add(button);
        add(west,BorderLayout.WEST);
        showResult = new JTextArea(10,10);
        showResult.setFont(new Font("宋体",Font.PLAIN,12));
        add(new JScrollPane(showResult),BorderLayout.CENTER);
```

```
        FindBookByXPath findBook;      //负责使用 XPath 查询图书的对象
        findBook =
        new FindBookByXPath(inputBookName,inputBookAuthor,inputBookISBN,inputPublish,
                            showResult,"book.xml");
        button.addActionListener(findBook);
        setBounds(10,10,900,300);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

FindBookByXPath.java

```
import javax.xml.xpath.*;
import org.w3c.dom.*;
import javax.swing.*;
import java.awt.event.*;
import org.xml.sax.*;
public class FindBookByXPath implements ActionListener {
    XPathFactory xPathFactory;
    XPath xPath;
    String positionPath;
    InputSource source;
    JTextField inputBookName,inputBookAuthor,inputBookISBN,inputPublish;
    JTextArea showResult;
    FindBookByXPath(JTextField inputBookName,JTextField inputBookAuthor,
                    JTextField inputBookISBN ,JTextField inputPublish,
                    JTextArea showResult,String fileName){
        this.inputBookName = inputBookName;
        this.inputBookAuthor = inputBookAuthor;
        this.inputBookISBN = inputBookISBN;
        this.inputPublish = inputPublish;
        this.showResult = showResult;
        xPathFactory = XPathFactory.newInstance();
        xPath = xPathFactory.newXPath();
        source = new InputSource(fileName);
        this.positionPath = positionPath;
    }
    public void actionPerformed(ActionEvent e){
        showResult.setText(null);
        String  bookName = inputBookName.getText().trim();
        System.out.println(bookName);
        String  authorName = inputBookAuthor.getText().trim();
        String  ID = inputBookISBN.getText().trim();
        String  publish = inputPublish.getText().trim();
        String predicates = "[contains(名称,'" + bookName + "') and " +
                            "contains(作者,'" + authorName + "') and " +
                            "contains(ISBN,'" + ID + "') and " +
                            "contains(出版社,'" + publish + "')]"; //谓词
        String path = "/图书列表/图书" + predicates + "/*";          //XPath 路径表达式
```

```
            System.out.println(path);
            try{
                NodeList nodelist =
                (NodeList)xPath.evaluate(path,source,XPathConstants.NODESET);
                int size = nodelist.getLength();
                for(int k = 0;k < size;k + + ){
                    Node node = nodelist.item(k);
                    String name = node.getNodeName();
                    showResult.append(" " + name + ":");
                    String content = node.getTextContent();
                    showResult.append(content + "\n");
                    if(name.startsWith("出版社"))
                        showResult.append("--------------------------\n");
                }
            }
            catch(Exception exp){
                showResult.setText(null);
                showResult.append("异常:" + exp);
            }
        }
    }
```

习 题 6

1. Element 类型节点对应 XML 中的标记还是标记中的文本？
2. 阅读下面的 XML 文件，回答问题。

Xiti2.xml

```
<教学楼>
   <教室 number = 'J301'>
       <桌子 number = 'T199'>
           <价格> 109 </价格>
       </桌子>
       <桌子 number = 'S102'>
           <价格> 360 </价格>
       </桌子>
   </教室>
   <教室 number = 'J502'>
       <桌子 number = 'S502'>
           <价格> 396 </价格>
       </桌子>
       <桌子 number = 'M202'>
           <价格> 267 </价格>
       </桌子>
   </教室>
</教学楼>
```

下列 XPath 路径表达式返回的节点集中有几个节点？这些节点都是什么类型？这

些节点分别对应着 XML 文件中的哪些标记或文本数据?

(1) /教学楼/教室[1]/桌子[1]/价格

(2) /教学楼/教室/桌子[contains(@number,'02') and 价格>= 360]

(3) /教学楼/教室/@*

(4) /教学楼/教室/@number/parent::教室

(5) /教学楼/教室[1]/桌子[2]/preceding::桌子/价格/text()

(6) /教学楼/教室[2]/*/*/text()

(7) /教学楼/教室[descendant::价格>=390]

(8) /教学楼/教室/桌子/价格[text()>=395]

3. 阅读下列文件:Xiti3.xml 和 JavaXPath.java,并写出 Java 程序中标注的【结果 1】至【结果 4】。

(1)【结果 1】:

(2)【结果 2】:

(3)【结果 3】:

(4)【结果 4】:

Xiti3.xml

```
<图书定单列表>
   <定单 ID="A1001">
       <图书名称>Java 课程设计</图书名称>
       <单价>20</单价>
       <订购数量>2</订购数量>
   </定单>
    <定单 ID="A1002">
       <图书名称>Java 设计模式</图书名称>
       <单价>30</单价>
       <订购数量>2</订购数量>
   </定单>
   <定单 ID="A1003">
       <图书名称>JSP 实用教程</图书名称>
       <单价>25</单价>
       <订购数量>2</订购数量>
   </定单>
</图书定单列表>
```

JavaXPath.java

```
import javax.xml.xpath.*;
import org.xml.sax.*;
import org.w3c.dom.*;
public class JavaXPath{
   public static void main(String args[]){
     try{ XPathFactory xPathFactory=XPathFactory.newInstance();
          XPath xPath=xPathFactory.newXPath();
          InputSource source=new InputSource("Xiti3.xml");
```

```
            String path = "/图书定单列表/定单[@ID = 'A1001']/单价";
            Double price =
            (Double)xPath.evaluate(path,source,XPathConstants.NUMBER);
            path = "/图书定单列表/定单[@ID = 'A1001']/订购数量";
            Double amount =
            (Double)xPath.evaluate(path,source,XPathConstants.NUMBER);
            double totalPrice = amount.doubleValue() * price.doubleValue();
            path = "/图书定单列表/定单[@ID = 'A1001']/@ID";
            Node id =
            (Node)xPath.evaluate(path,source,XPathConstants.NODE);
            String idNumber = id.getNodeValue();
            System.out.print(idNumber + "定单的总额: ");   //【结果 1】
            System.out.println(totalPrice);               //【结果 2】
            String pathOne = "/图书定单列表/定单[contains(图书名称,'Java')]/单价";
            String pathTwo = "/图书定单列表/定单[contains(图书名称,'Java')]/订购数量";
            NodeList nodelist =
            (NodeList)xPath.evaluate(pathOne + "|" + pathTwo,
                                    source,XPathConstants.NODESET);
            int size = nodelist.getLength();
            System.out.println(size);                     //【结果 3】
            double priceSum = 0;
            int m = 0;
            while(m < size){
               Node node = nodelist.item(m);
               double bookPrice = Double.parseDouble(node.getTextContent().trim());
               node = nodelist.item(m + 1);
               double bookAmount = Double.parseDouble(node.getTextContent().trim());
               priceSum = priceSum + bookPrice * bookAmount;
               m = m + 2;
            }
            System.out.println(priceSum);                 //【结果 4】
        }
        catch(Exception exp){
            System.out.println(exp);
        }
    }
}
```

4. 有如下的 XML 文件：Xiti4.xml，请编写 Java 程序（参考例 10），计算出“货品列表”中全部货品的总重量。

Xiti4.xml

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<货品列表>
   <货品>
      <名称> 电视机</名称>
      <重量> 23.8 </重量>
   </货品>
  <货品>
```

```
        <姓名> 洗衣机</名称>
        <重量>67.2</重量>
    </货品>
      <货品>
        <名称> 冰箱</名称>
        <重量>89.8</重量>
    </货品>
</货品列表>
```

5. 参照例 11 编写一个学生基本信息查询系统。

第7章 XML与数据库

主要内容

- JDBC
- Microsoft Access 数据库
- 连接数据库
- XML 至数据库
- 数据库至 XML

许多应用程序都使用数据库来管理、存储数据，尽管数据库在数据查询、修改、保存和安全等方面有着其他数据处理手段无法替代的地位，但随着网络的迅速发展，让各种应用程序方便地交互各自数据库中的数据显得越来越重要。不同数据库之间因为数据格式和版本的不同，以及系统设计上的限制，使得不同数据库之间很难实现方便地交互数据。

XML 不仅能使应用程序方便地组织数据的结构，而且各种应用程序通过使用 XML 文件可以方便地交互它们之间的数据。因此，一个应用程序可能需要将数据库中的某些数据转化为 XML 文件，以便与其他程序交互这些数据。

本章主要从以下两个方面来讲解。

- 一个应用系统可能需要将数据库表中的某些记录转化为一个 XML 文件，以便与其他系统交互数据，发挥 XML 文件在数据交换上的优势(如图 7.1 所示)。
- 系统获得一个 XML 文件后，可能需要将 XML 文件某些标记中的内容转化为数据库中表的一条记录，以便发挥数据库在管理数据方面的优势(如图 7.1 所示)。

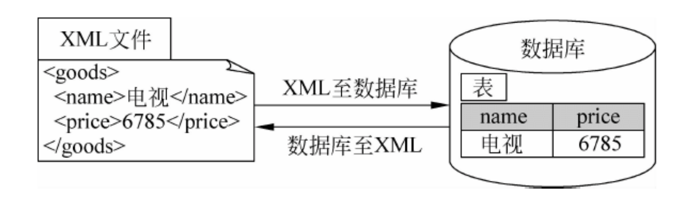


图 7.1 XML 中数据与数据库中记录的相互转化

本章不讲解数据库，因此要求读者学习过数据库的一些基本知识，了解(关系)数据库是以表的形式来组织数据的，而表是记录的集合。

本章主要讲解如何将 XML 文件的若干标记中的内容转化为数据库中表的一条记

录，以及如何将数据库中表的若干条记录转化为一个 XML 文件。

为满足本章的需要，在 7.1 节介绍有关 JDBC 的知识，以便编写和数据库交互信息的 Java 应用程序。

7.1 JDBC

JDBC(Java DataBase Connectivity)提供了访问数据库的 API，即由一些 Java 类和接口组成，是 Java 运行平台的核心类库中的一部分。使用 JDBC 可以实现对数据库中表的记录的查询、修改和删除等操作。JDBC 技术在数据库开发中占有很重要的地位，它操作不同的数据库仅仅在连接方式上有差异，JDBC 的应用程序一旦和数据库建立了连接，就可以使用 JDBC 提供的 API 操作数据库，如图 7.2 所示。

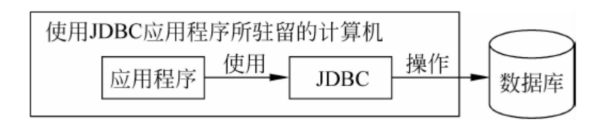


图 7.2　使用 JDBC 操作数据库

7.2 Microsoft Access 数据库

JDBC 操作不同的数据库仅仅是连接方式上的不同，为了便于教学，使用 Microsoft Access 来讲解 JDBC。一方面，考虑到许多学校的实验环境都是 Windows 2000/XP 系统，在安装 Office 的同时就会默认地安装 Microsoft Access 数据库管理系统。另一方面，Microsoft Access 也是常用的数据库管理系统之一，具有速度快、使用方便等特点。

表是关系型数据库的基本单位，由于本章不是讲解数据库设计，因此本节只是简单介绍如何使用 Microsoft Access 数据库管理系统创建数据库和在数据库中创建表。

7.2.1 建立数据库

单击 Windows 2000/XP 系统桌面上的“开始”→“(所有)程序”→ “Microsoft Office” “Microsoft Access”选项启动数据库管理系统，然后选择新建数据库，如图 7.3 所示。



图 7.3　新建数据库

将新建的数据库命名为 employee.mdb，并保存到磁盘中，例如保存到：C:\chapter7。

7.2.2 创建表

创建好数据库后，就可以在该数据库下建立若干个表。打开 employee.mdb 数据库，在选择界面上选择“使用设计器创建表”选项后，单击“设计”按钮，将出现创建表的界面，使用该界面创建名字为 person 的表，并指定字段(列)以及字段的数据类型，如图 7.4 所示。



图 7.4 创建表

7.3 连接数据库

应用程序为了能和数据库交互信息，必须首先和数据库建立连接。使用 JDBC 的应用程序无论采用哪种方式连接数据库，都不会影响操作数据库的逻辑代码，这非常有利于代码的维护和升级。本章的 XML 文件都需要和数据库建立连接，所涉及的和数据库交互信息的代码不依赖数据库的连接方式，因此本节介绍常用的连接方式之一——JDBC-ODBC 桥接器。

7.3.1 JDBC-ODBC 桥接器

应用程序和数据库建立连接的一种常见方式是使用 JDBC-ODBC 桥接器。使用 JDBC-ODBC 桥接器方式的机制是，应用程序只需建立 JDBC 和 ODBC 之间的连接，即建立 JDBC-ODBC 桥接器，而和数据库的连接由 ODBC 去完成。

JDBC-ODBC 桥接器的优点是：ODBC(Open DataBase Connectivity)是 Microsoft 引进的数据库连接技术，提供了数据库访问的通用平台，而且 ODBC 驱动程序被广泛地使用，建立这种桥接器后，使得 JDBC 有能力访问几乎所有类型的数据库。缺点是：使得应用程序依赖于 ODBC，移植性较差，也就是说，应用程序所驻留的计算机必须提供 ODBC。

应用程序负责使用 JDBC 提供的 API 建立 JDBC-ODBC 桥接器，然后应用程序就可以请求和数据库建立连接，连接工作由 ODBC 完成。

需要强调是，ODBC 使用“数据源”来管理数据库，所以必须事先将某个数据库设置成 ODBC 所管理的一个数据源，应用程序只能和 ODBC 管理的数据源建立连接。使用 JDBC-ODBC 桥接器方式和数据库建立连接如图 7.5 所示。

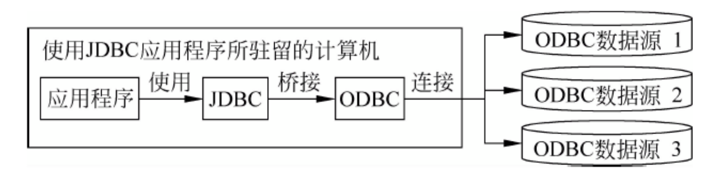


图 7.5 JDBC-ODBC 桥接器

7.3.2 ODBC 数据源

若应用程序要与 7.2 节中的 employee.mdb 数据库建立连接,则需要将 employee.mdb 数据库设置成一个 ODBC 数据源。应用程序所在的计算机负责创建数据源,即将本地或远程机上的数据库设置成应用程序要访问的数据源。因此,必须保证应用程序所在计算机有 ODBC 系统(Windows 2000、Windows XP 都有 ODBC 系统)。创建 ODBC 数据源的操作步骤如下。

1. 创建、修改或删除数据源

选择"控制面板"→"管理工具"→"ODBC 数据源"(某些 Windows XP 系统,需选择"控制面板"→"性能和维护"→"管理工具"→"ODBC 数据源")选项。双击"ODBC 数据源"图标,出现"ODBC 数据源管理器"对话框,如图 7.6 所示,该对话框显示了用户已有的数据源的名称。选择"用户 DSN"选项,单击"添加"按钮,可以创建新的数据源;单击"配置"按钮,可以重新配置已有的数据源;单击"删除"按钮,可以删除已有的数据源。



图 7.6 数据源管理器

2. 为数据源选择驱动程序

在"ODBC 数据源管理器"对话框(图 7.6 所示的界面)中单击"添加"按钮,出现"创建新数据源"对话框,如图 7.7 所示。在该对话框中为新增的数据源选择驱动程序,因为要访问 Microsoft Access 数据库,所以选择 Microsoft Access Driver(*.mdb),单击"完成"按钮。



图 7.7 为数据源选择驱动程序

3. 设置数据源名称及数据库所在位置

在“创建新数据源”对话框(如图 7.7 所示)中单击“完成”按钮将出现“ODBC Microsoft Access 安装”对话框(如图 7.8 所示)。在该对话框中为数据源起一个你喜欢的名字,如 company,这个数据源就是指某个数据库。最后单击该对话框“数据库”选项组中的“选择”按钮将数据库设置成第 7.2 节所建立的 employee.mdb 数据库即可。



图 7.8 设置数据源的名称和数据库所在位置

注意:如果设置数据源时,提示“非法路径”,请关闭 Microsoft Access 打开的 employee.mdb 数据库。

7.3.3 建立连接

编写连接数据库代码时不会出现数据库的名称,只会出现数据源的名字。

应用程序和数据源(即数据源所代表的数据库)建立连接的步骤如下。

1. 建立 JDBC-ODBC 桥接器

JDBC 使用 java.lang 包中的 Class 类建立 JDBC-ODBC 桥接器。Class 类通过调用它的静态方法 forName 加载 sun.jdbc.odbc 包中的 JdbcOdbcDriver 类建立 JDBC-ODBC 桥接器。建立桥接器时可能发生异常,必须捕获这个异常,建立桥接器的代码是:

```
try {   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch(ClassNotFoundException e){
      System.out.println(e);
}
```

2. 和数据库建立连接

首先使用 java.sql 包中的 Connection 类声明一个对象,然后再使用 DriverManager 类调用它的静态方法 getConnection 创建连接对象:

```
Connection con =
DriverManager.getConnection("jdbc:odbc:数据源名","login name","password");
```

倘若没有为数据源设置 login name 和 password,那么连接形式是:

```
Connection con = DriverManager. getConnection("jdbc:odbc:数据源名字", "","");
```

为了能和数据源 company 交换数据,建立连接时应捕获 SQLException 异常:

```
try{ Connection con = DriverManager.getConnection("jdbc:odbc:company", "","");
```

```
}
catch(SQLException e){
     System.out.println(e);
}
```

应用程序一旦和某个数据源建立连接，就可以通过 SQL 语句和该数据源所指定的数据库中的表交互信息，例如查询、修改、更新表中的记录。

下面的例 1 是一个 Java 应用程序，负责连接到数据源 company 代表的数据库，向数据库的 person 表添加记录，并查询 person 表中的全部记录，InsertAndQuery.java 的运行效果如图 7.9 所示。

【例 1】

InsertAndQuery.java

```
a1004,张三,1995-12-12,3000.0
a1005,李四,1996-09-10,5000.0
```

图 7.9 添加与查询记录

```
import java.sql.*;
public class InsertAndQuery{
    public static void main(String args[]){
         Connection con;
         Statement sql;
         ResultSet rs;
         try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         }
         catch(ClassNotFoundException e){
               System.out.println(""+e);
         }
         try{  con=DriverManager.getConnection("jdbc:odbc:company","","");
               sql=con.createStatement();
               sql.executeUpdate
               ("INSERT INTO person VALUES('a1004','张三','1995-12-12',3000)");
               sql.executeUpdate
               ("INSERT INTO person VALUES('a1005','李四','1996-09-10',5000)");
               rs=sql.executeQuery("SELECT * FROM person");
               while(rs.next()){
                   String number=rs.getString(1);
                   String name=rs.getString(2);
                   Date birth=rs.getDate(3);
                   double salary=rs.getDouble(4);
                   System.out.println(number+","+name+","+birth+","+salary);
               }
               con.close();
         }
         catch(SQLException e){
               System.out.println(e);
         }
    }
}
```

7.4 XML 至数据库

本节讲解如何将 XML 文件的某些标记中的内容转化为数据库中表的一条记录，其主要步骤如下。

1. 获取标记中的数据

如果要将某些标记中的内容转化为数据库中表的一条记录,那么可以使用 DOM 解析器、SAX 解析器或 XPath 路径表达式获取这些标记中的文本。例如,这些标记的名称是“雇员号”、“姓名”、“出生日期”和“薪水”,那么获取的这些标记中的文本则分别是:“A1006”、“王经路”、“1995-11-11”和“6789”。

2. 将获取的数据作为一条记录添加到数据库

连接数据库,将获取的数据作为一条记录添加到数据库中,例如:

```
"INSERT INTO person VALUES('A1006','王经路','1995-11-11',6789)";
```

例 2 中有一个 XML 文件 exmaple7_2.xml 和一个应用程序 XMLToDatabae.java,应用程序将 XML 文件中标记名称为“雇员号”、“姓名”、“出生日期”和“薪水”的文本作为一条记录添加到数据库的 person 表中(见 7.2 节创建的 employee.mdb)。运行例 2 中的 Java 程序后,用 Microsoft Access 数据库管理系统打开 employee.mdb 数据库中的 person 表,可以看到该表中新添加的记录,效果如图 7.10 所示。

person : 表

number	name	birthday	salary
a1004	张三	1995-12-12	3000
a1005	李四	1996-9-10	5000
A1006	王经路	1995-11-11	6789
A1007	赵懂知	1995-6-28	5673
			0

图 7.10 将 XML 标记中的数据转化为记录

【例 2】

example7_2.xml

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<雇员列表>
   <雇员>
      <雇员号> A1006 </雇员号>
      <姓名>王经路</姓名>
      <出生日期> 1995 - 11 - 11 </出生日期>
      <薪水> 6789 </薪水>
   </雇员>
   <雇员>
      <雇员号> A1007 </雇员号>
      <姓名>赵懂知</姓名>
      <出生日期> 1995 - 06 - 28 </出生日期>
      <薪水> 5673 </薪水>
   </雇员>
</雇员列表>
```

XMLToDatabase.java

```
import javax.xml.xpath.* ;
import org.xml.sax.* ;
import org.w3c.dom.* ;
```

```
import java.sql.*;
public class XMLToDatabase{
   public static void main(String args[]){
     Connection con = null;
     Statement sql = null;
     ResultSet rs = null;
     try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     }
     catch(Exception e){
            System.out.println("" + e);
     }
     try{ con = DriverManager.getConnection("jdbc:odbc:company","","");
          sql = con.createStatement();
          XPathFactory xPathFactory = XPathFactory.newInstance();
          XPath xPath = xPathFactory.newXPath();
          InputSource source = new InputSource("example9_2.xml");
          String path = "/雇员列表/雇员";
          NodeList nodeList = (NodeList) xPath. evaluate (path, source, XPathConstants.
NODESET);
          int size = nodeList.getLength();
          for(int i = 0;i < size;i + + ){
              int m = i + 1;
              path = "/雇员列表/雇员[" + m + "]/*/text()";
              nodeList = (NodeList)xPath.evaluate(path,source,XPathConstants.NODESET);
              int length = nodeList.getLength();
              String [] a = new String[length];
              for(int k = 0;k < length;k + + ){
                  Node node = nodeList.item(k);
                  a[k] = node.getTextContent().trim();
              }
              String insertData =
              "INSERT INTO person VALUES('" + a[0] + "','" + a[1] + "','" + a[2] + "'," + a[3]
+ ")";
               sql.executeUpdate(insertData);
          }
          con.close();
     }
     catch(Exception exp){
          System.out.println(exp);
     }
   }
}
```

7.5 数据库至 XML

一个应用系统可能需要将数据库表中的某些记录转化为一个 XML 文件,以便与其他系统交互数据,发挥 XML 文件在数据交换上的优势,其主要步骤如下。

1. 查询记录

连接数据库查询记录(参见 7.2 节的 employee.mdb),例如,查询到记录:

```
('a1005','李四','1996-09-10',5000)
```

2. 将记录中的字段值作为标记的内容

在内存中创建一个 Document 对象,在其中建立名称分别为“雇员号”、“姓名”、“出生日期”和“薪水”的节点,并将步骤 1 查询到的记录中各个字段(列)的值分别作为“雇员号”、“姓名”、“出生日期”和“薪水”节点中的文本内容,然后使用 DOM 生成一个 XML 文件(参见第 4 章的 4.10 节)。

下面例 3 中的应用程序 DatabaseToXML 负责创建一个 XML 文件 newXML.xml,并将数据库中的全部记录分别作为 XML 文件 newXML.xml 中标记名称为“雇员号”、“姓名”、“出生日期”和“薪水”中的文本。在运行例 3 中的 Java 程序之后,用浏览器打开 newXML.xml 的效果如图 7.11 所示。

```
地址(D)  E:\chapter7\newXML.xml

  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
- <雇员列表>
  - <雇员>
      <雇员号>a1004</雇员号>
      <姓名>张三</姓名>
      <出生日期>1995-12-12</出生日期>
      <薪水>3000.0</薪水>
    </雇员>
  - <雇员>
      <雇员号>a1005</雇员号>
      <姓名>李四</姓名>
      <出生日期>1996-09-10</出生日期>
      <薪水>5000.0</薪水>
    </雇员>
  - <雇员>
      <雇员号>A1006</雇员号>
      <姓名>王经路</姓名>
      <出生日期>1995-11-11</出生日期>
      <薪水>6789.0</薪水>
    </雇员>
  - <雇员>
      <雇员号>A1007</雇员号>
      <姓名>赵懂知</姓名>
      <出生日期>1995-06-28</出生日期>
      <薪水>5673.0</薪水>
    </雇员>
  </雇员列表>
```

图 7.11　将数据库中的表转化为 XML 文件

【例 3】

DatabaseToXML.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.dom.*;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
```

```
import java.sql.*;
public class DatabaseToXML{
   public static void main(String args[]){
      Connection con;
      Statement sql;
      ResultSet rs;
      String [] number = {""};
      String [] name = {""};
      String [] date = {""};
      String [] salary = {""};
      try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      }
      catch(ClassNotFoundException e){
            System.out.println("" + e);
      }
      try{ con = DriverManager.getConnection("jdbc:odbc:company","","");
           sql = con.createStatement
           (ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
           rs = sql.executeQuery("SELECT * FROM person");
           rs.last();
           int recordAmount = rs.getRow();
           number = new String[recordAmount];
           name = new String[recordAmount];
           date = new String[recordAmount];
           salary = new String[recordAmount];
           int k = 0;
           rs.beforeFirst();
           while(rs.next()){
              number[k] = rs.getString(1);
              name[k] = rs.getString(2);
              date[k] = rs.getDate(3).toString();
              salary[k] = String.valueOf(rs.getDouble(4));
              k++;
           }
           con.close();
        }
        catch(SQLException e){
              System.out.println(e);
        }
        try{ DocumentBuilderFactory factory =
              DocumentBuilderFactory.newInstance();
              DocumentBuilder domPaser = factory.newDocumentBuilder();
              Document document = domPaser.newDocument();
              document.setXmlVersion("1.0");
              Element root = document.createElement("雇员列表");
              document.appendChild(root);
              for(int k = 0;k < name.length;k++){
                 Node employee = document.createElement("雇员");
                 root.appendChild(employee);
```

```
                Node xuehao = document.createElement("雇员号");
                xuehao.appendChild(document.createTextNode(number[k]));
                Node xingming = document.createElement("姓名");
                xingming.appendChild(document.createTextNode(name[k]));
                Node chusheng = document.createElement("出生日期");
                chusheng.appendChild(document.createTextNode(date[k]));
                Node pay = document.createElement("薪水");
                pay.appendChild(document.createTextNode(salary[k]));
                employee.appendChild(xuehao);
                employee.appendChild(xingming);
                employee.appendChild(chusheng);
                employee.appendChild(pay);
            }
            TransformerFactory transFactory = TransformerFactory.newInstance();
            Transformer transformer = transFactory.newTransformer();
            DOMSource domSource = new DOMSource(document);
            File f = new File("newXML.xml");
            FileOutputStream out = new FileOutputStream(f);
            StreamResult xmlResult = new StreamResult(out);
            transformer.transform(domSource,xmlResult);
            out.close();
        }
        catch(Exception e){
            System.out.println(e);
        }
    }
}
```

习 题 7

1. 参考例 2，将 XML 文件中的某些标记的内容作为一条记录添加到数据库中。
2. 参考例 3，将 employee 数据库 person 表中的前 3 条记录转化为一个 XML 文件。

第 8 章 XML 与 CSS

主要内容

- XML 关联 CSS
- 标记与样式表
- 设置文本的显示方式
- 字体
- 文本样式
- 边框
- 颜色和背景
- 显示图像
- 设置鼠标的形状

XML 关心的是数据的结构，并能很好、方便地描述数据，但它不提供数据的显示功能。因此，浏览器不能直接显示 XML 文件中标记的文本内容，如果想让浏览器显示 XML 文件中标记的文本内容，必须以某种方式告诉浏览器如何显示。W3C 为 XML 数据显示发布了两个规范：CSS(层叠样式表)和 XSL(可扩展样式语言)。当 XML 文件和 CSS 文件或 XSL 文件相关联后，浏览器将按照 CSS 文件或 XSL 文件给出的显示方式来显示 XML 文件中标记的文本内容(如图 8.1 所示)。

XML文件 —关联→ CSS文件

图 8.1 使 XML 和 CSS 相关联

XSL 不仅能帮助 XML 显示数据，而且本身还有许多应用领域，对 XSL 的讲解已超出本书范围，本章重点讲述如何使用 CSS 显示 XML 标记中的文本内容。CSS 能为 XML 提供数据显示的外观，且语法简单，便于学习使用，在某些应用中，使用 CSS 可以为 XML 提供很好的数据显示外观。

8.1 初识 CSS

在 CSS 中，最重要的概念就是样式表。样式表是一组规则，通过这组规则告诉浏览器用什么样式来显示文本，例如，告诉浏览器使用什么样的字体、颜色和页边距来显示文本。一个 CSS 文件就是由若干个样式表构成的文本文件，因此编写 CSS 文件的关键就是

编写其中的样式表。

样式表的格式如下：

```
样式表名称
{
   样式规则
}
```

编写 CSS 文件时，样式表中的“样式表名称”是标记的名称（有关细节见后面的 8.3.1 小节）。样式表中的“样式规则”是若干个用分号分隔的“属性名：属性值”，例如：

```
name
{  display:block;font - size:12pt;font - weight:bold;
}
```

就是 CSS 中的一个样式表，该样式表用来显示 XML 文件中 name 标记包含的文本内容。其中的“display:block;”告知浏览器将标记“<name>…</name>”中的文本内容显示在一个“块区域”；“font-size:12pt;”能使得所显示的文本的字体的大小是 12 镑（小 4 号字）；“font-weight:bold”的作用是使文本用粗体字显示。

一个层叠样式表（CSS）就是由若干个样式表构成的文本文件，该文本文件可以使用“ANSI”或“UTF-8”编码来保存，文件的扩展名是“.css”。

注意：样式表定义中的“样式表名称”不要含有非 ASCII 字符，目前的大部分浏览器不支持这样的样式表。

8.2 XML 关联 CSS

为了让 XML 使用层叠样式表，XML 文件必须使用操作指令：

```
<?xml - stylesheet  href = "样式表的 URI"  type = "text/css" ?>
```

将当前 XML 文件关联到某个层叠样式表。操作指令中 href 属性指定的属性值中的 URI 如果是一个文件的名字，该文件必须和 XML 文件在同一目录中，例如：

```
<?xml - stylesheet  href = "show.css"  type = "text/css" ?>
```

如果操作指令中 href 属性指定的属性值中的 URI 是一个 URL，该 URL 必须是有效的（可访问的），例如：

```
<?xml - stylesheet  href = "http://www.yahoo.com/show.css"  type = "text/css" ?>
```

当 XML 文件和一个 CSS 文件相关联后，如果用浏览器打开 XML 文件（XML 必须是规范的），浏览器不再显示 XML 源文件，而是使用 CSS 文件中的样式表显示 XML 文件中标记包含的文本数据。

在 XML 中，操作指令写在 XML 文件根标记之前，用

```
<?操作指令的名字 操作内容的文本描述
```

开始,用

```
?>
```

结束的指令。例如,操作指令:

```
<?xml-stylesheet href = "样式表的 URI" type = "text/css" ?>
```

将一个 XML 文件与 CSS 样式表相关联。

下面的例 1 中有一个 XML 文件 example8_1.xml 和一个 CSS 文件 oneCSS.css,example8_1.xml 和 oneCSS.css 相关联,并保存在相同的目录中。用浏览器打开 example8_1.xml 的效果如图 8.2 所示。

【例 1】

oneCSS.css

```
name
{ display:block;font-size:18pt;font-weight:bold
}
sex
{ display:line;font-size:12pt;font-style:italic
}
birthday
{ display:line;font-size:10pt;font-weight:bold
}
```

图 8.2 打开和 CSS 关联的 XML 文件

example8_1.xml

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<?xml-stylesheet href = "oneCSS.css" type = "text/css" ?>
<student>
   <name>李小林
      <sex>男 </sex>
      <birthday>1998 年 12 月 22 日 </birthday>
   </name>
   <name>金锦巾
      <sex>女 </sex>
      <birthday>1999 年 08 月 10 日 </birthday>
   </name>
</student>
```

8.3 标记与样式表

CSS 中的样式表负责给出 XML 文件中标记包含的文本数据的显示外观,为此,样式表的名称必须和 XML 文件中标记的名称建立某种联系。本节简要介绍如何为样式表命名以及 CSS 的显示规则,本节之后的内容将详细讲解如何编写各类样式表。

8.3.1 标记的名字与样式表的名称

对于 XML 文件,样式表中的"样式表名称"可以是标记的名称,也可以是标记的名称

与该标记的 ID 属性值用“#”连接起来的字符串。

当 XML 的许多标记具有相同的名字，并准备使用不同的外观来显示它们的内容时，“样式表名称”使用标记的名称与该标记的 ID 属性值用“#”连接起来的字符串就显得很方便，因为可以为那些名字相同的标记指定不同的 ID 属性值。例如，假设 XML 文件中有两个标记的名称都是 time，XML 文件可以为这两个名字相同的 time 标记指定不同的 ID 属性值：

```
<time  ID="a01">2010-12-12</time>
<time  ID="a02">2010-10-10</time>
```

然后让对应的 CSS 文件中含有如下的两个样式表：

```
time#a001
{  display:block;font-size:19pt;font-weight:bold;
}
time#a002
{  display:line;font-size:16pt;font-weight:bold;
}
```

如果有多个标记的内容需要用完全一样的外观来显示，“样式表名称”也可以是这些标记的名称用逗号分隔的字符串。例如，XML 文件中名称是 name、sex 和 birthday 的标记想用同样的外观显示各自内容中的文本数据，那么对应的 CSS 文件中，就可以有如下的样式表：

```
name,sex,birthday
{  display:block;font-size:36pt;font-weight:bold;
}
```

8.3.2 CSS 的显示规则

当用浏览器打开 XML 文件时，浏览器将按照标记在 XML 文件中出现的顺序，并用该标记在 CSS 中对应的样式表显示该标记包含的文本数据，如果标记在 CSS 中没有对应的样式表，浏览器将使用默认的显示规则显示该标记中的文本数据。

下面的例 2 中的 XML 文件 example8_2.xml 和 CSS 样式表 twoCSS.css 相关联。

【例 2】

twoCSS.css

```
name
{  display:block;font-size:18pt;font-weight:bold
}
price
{  display:line;font-size:16pt;font-style:italic
}
madeTime
{  display:line;font-size:9pt;font-weight:bold
}
```

example8_2. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<?xml - stylesheet href = "twoCSS.css" type = "text/css" ?>
<goods>商品列表:
     <name>
       诺基亚手机
       <price>1907 元/部 </price>
       <madeTime>2011.09.28 </madeTime>
   </name>
   <name>
       苹果 IPad
       <price>5668 元/台 </price>
       <madeTime>2011.10.10 </madeTime>
   </name>
</goods>
```

例 2 中的 XML 文件使用 twoCSS. css 层叠样式表显示标记包含的文本数据，那么显示文本数据的顺序是：

- 显示根标记包含的文本“商品列表”，使用默认规则(因为 twoCSS. css 中没有对应根标记的样式表)。
- 显示第一个 name 标记包含的文本数据“诺基亚手机”，所用样式表是 twoCSS. css 中名称是 name 的样式表。
- 显示第一个 name 标记的 price 和 madeTime 子标记包含的文本数据“1907 元/部”和“2010. 09. 28”，所用样式表分别是 twoCSS. css 中名称是 price 和 madeTime 的样式表。
- 显示第二个 name 标记包含的文本数据“苹果 IPad”，所用样式表是 twoCSS. css 中名称是 name 的样式表。
- 显示第二个 name 标记的 price 和 madeTime 子标记包含的文本数据“5668 元/台”和“2010. 10. 10”，所用样式表分别是 twoCSS. css 中名称是 price 和 madeTime 的样式表。

用浏览器打开 example8_2. xml 的效果如图 8. 3 所示。



图 8. 3　CSS 按标记顺序显示数据

8. 4　数据结构与显示相分离

在第 1 章曾介绍了 HTML 与 XML 的不同。HTML 是设计数据显示外观的语言，它的核心是把数据和数据的显示外观捆绑在一起，即 HTML 中的标记的出发点不是为

了体现数据的含义，而是体现数据的显示格式，整个 HTML 文件的目的不是为了组织数据的结构，而是为了组织数据的显示外观；XML 的核心是描述数据的组织结构，其标记名称是对标记所包含的数据内容的抽象，而不是数据的显示格式，XML 文件通过使用若干个标记来表示数据的组织结构，通过和 CSS 或 XSL 相关联来显示标记中的文本数据，从而有效地分离了数据的组织结构和显示外观。

以下通过一个简单的例子来体会 XML 通过和 CSS 关联来有效地分离数据的组织结构和显示外观。在下面的例 3 中，XML 文件 example8_3. xml 和 threeCSS. css 相关联。如果在 example8_3. xml 中再增加若干个 shop 标记，却不修改 threeCSS. css 文件，新增的 shop 标记包含的文本内容就可以被浏览器显示；另一方面，如果改了 threeCSS. css 中的样式表，不修改 XML 文件，浏览器就会用新的样式表显示标记包含的文本数据。用浏览器打开 example8_3. xml 的效果如图 8.4 所示。



图 8.4 使用 CSS 显示 XML 中的数据

【例 3】

threeCSS. css

```
shop#a001
{  display:block;font-size:18pt;font-weight:bold; color:red;
}
shop
{  display:block;font-size:18pt;font-weight:bold;
}
startTime
{  display:line;font-size:15pt;font-weight:bold;color:rgb(100,129,70);
}
endTime
{  display:line;font-size:15pt;font-weight:bold;color:rgb(0,129,170);
}
```

example8_3. xml

```
<?xml  version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet href="threeCSS.css" type="text/css" ?>
<businessTime>
   <shop ID="a001">
       <name>体育商城</name>
       <startTime>开门时间: 08:30</startTime>
       <endTime>关门时间: 18:30</endTime>
   </shop>
   <shop>
       <name>美音琴行</name>
```

```
        <startTime>开门时间: 07:30</startTime>
        <endTime>关门时间: 22:30</endTime>
    </shop>
</businessTime>
```

注意:浏览器把"英文输入法"状态编辑输入的连续空格按一个空格显示,如果准备显示标记包含的全部空格,在编辑 XML 文件时,需将输入法切换成"中文输入法",并选择"全角状态"来编辑空格字符。

8.5 设置文本的显示方式

样式表中通过指定属性 display 的值来设置文本的显示方式。

8.5.1 块方式

在样式表中的样式规则中通过将属性 display 的值指定为 block,使得文本在浏览器的一个块区域中显示。例如:

```
name
{   display:block;
}
```

上述 name 样式表使用块区域显示文本,块的位置默认为靠左对齐,块的大小依赖于需要显示的文本中字符的数量、字符的大小以及当前浏览器显示区域的大小。

可以通过进一步设置 width 和 height 属性的值来准确地设置块区域的宽度和高度,例如:

```
name
{   display:block;width=280;height=180
}
```

在 8.2.2 小节中讲述了 CSS 显示标记包含的文本内容的顺序是按照标记在 XML 文件中出现的顺序,并用该标记在 CSS 中对应的样式表显示该标记包含的文本数据,其特点如下。

1. 同级别标记

如果有若干同级别的标记所对应的样式表都是使用块区域显示文本,那么这些块区域将按照标记的先后顺序靠左对齐。

2. 子标记

如果样式表指定某个标记的显示方式是 block,而另外一个样式表为当前标记的子标记指定的显示方式也是 block,那么为子标记指定的块区域将被包含在为父标记指定的块区域中。

可以通过设置 position 属性的值为 absolution 改变块区域默认靠左对齐的方式,但必须要同时设置 left、top、width 和 height 属性的值,以便准确地设置块区域的位置和大

小，下面的样式表设置了 position、left、top、width 和 height 属性的值：

```
name
{  display:block;
   position:absolution;
   left = 100; top = 120;
   width = 200; height = 120;
}
```

在下面的例 4 中，三个标记以及子标记的内容的显示方式都是块方式，浏览器显示 example8_4.xml 的效果如图 8.5 所示。

第一教学楼(靠左对齐)。 我是第一教学楼的1101教室。 我是第一教学楼的1201教室。
第二教学楼(靠左对齐)。 我是第二教学楼的2101教室。 我是第二教学楼的2201教室。

综合办公楼,设置了准确位置。

图 8.5 样式表使用块区域显示文本

【例 4】

fourCSS.css

```
name
{    display:block;
     background - color:cyan;
}
classRoom
{    display:bolck;
     background - color:rgb(200,210,220);
}
name # special
{    display:block;
     position:absolution;
     width = 150;
     height = 60;
     left = 180;
     top = 95;
     background - color:pink
}
```

example8_4.xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<?xml - stylesheet href = "fourCSS.css" type = "text/css" ?>
< house >
  < name >
    第一教学楼(靠左对齐).
    < classRoom > 我是第一教学楼的 1101 教室.</classRoom >
    < classRoom > 我是第一教学楼的 1201 教室.</classRoom >
  </name >
```

```
    <name>
       第二教学楼(靠左对齐).
       <classRoom>我是第二教学楼的 2101 教室.</classRoom>
       <classRoom>我是第二教学楼的 2201 教室.</classRoom>
    </name>
    <name ID="special">
       综合办公楼,设置了准确位置.
    </name>
</house>
```

8.5.2 行方式

在样式表中的样式规则中通过将属性 display 的值指定为 line,使得文本以行的方式在浏览器中显示。例如:

```
name
{  display:line;
}
```

在第 8.2.2 小节中讲述了标记中内容的显示顺序是按照标记在 XML 文件中出现的顺序,并用该标记在 CSS 中对应的样式表显示该标记中的文本数据,其特点如下。

1. 同级别标记

如果有若干同级别的标记所对应的样式表都是使用行方式来显示文本,那么这些行将按照标记的先后顺序首尾相接。

2. 子标记

如果样式表为某个标记指定的显示方式是 block,而另外一个样式表为当前标记的子标记指定的显示方式是 line,那么为子标记指定的行将被包含在为父标记指定的块区域中。

可以通过设置 position 属性的值为 absolution 改变某行默认地按先后顺序尾随在另一行之后的方式,但必须要同时设置 left、top 属性的值,以便准确地设置行的位置,下面的样式表设置了 position、left 和 top 属性的值:

```
name
{  display:line;
   position:absolution;
   left=100; top=120;
}
```

在下面的例 5 中,student 标记的显示方式是 block,它的两个子标记的显示方式是 line。浏览器显示 example8_5.xml 的效果如图 8.6 所示。

【例 5】

fiveCSS.css

```
student
```

张三 男 1996-12-12
赵花 女 1998-11-11

图 8.6 块中的行方式

```
{   display:block;
    font-size:18pt;
    background-color:cyan;
}
sex
{  display:line;
   font-size:12pt;
   color:red;
}
name
{  display:line;
   font-size:16pt;
   color:green;
}
birth
{  display:line;
   font-size:10pt;
}
```

example8_5.xml

```
<?xml  version="1.0"  encoding="UTF-8" ?>
<?xml-stylesheet href="fiveCSS.css" type="text/css" ?>
<people>
  <student>
      <name>张三 </name>
      <sex>男 </sex>
      <birth>1996-12-12</birth>
  </student>
  <student>
      <name>赵花 </name>
      <sex>女 </sex>
      <birth>1998-11-11</birth>
  </student>
</people>
```

8.5.3 按列表方式

在样式表的样式规则中通过将属性 display 的值指定为 list-item，使得文本按列表方式在浏览器中显示。例如：

```
name
{  dislpaly:list-item;
}
```

在 8.2.2 小节中我们讲述了标记中内容的显示顺序是按照标记在 XML 文件中出现的顺序，并用该标记在 CSS 中对应的样式表显示该标记中的文本数据，其特点如下。

1. 同级别标记

如果有若干同级别的标记对应的样式表都使用列表来显示文本,那么这些列表将按照标记的先后独行显示,而且列表的项目符号会按先后顺序递增。

2. 子标记

如果样式表为某个标记指定的显示方式是 block,而另外一个样式表为当前标记的子标记指定的显示方式是 list-item,那么为子标记指定的列表将被包含在为父标记指定的块区域中。如果样式表为某个标记指定的显示方式是 list-item,而另外一个样式表为当前标记的子标记指定的显示方式也是 list-item,那么为子标记指定的列表将成为为父标记指定的列表的子列表。

和 list-item 属性有关的属性是 list-style-type,也就说 list-style-type 属性可以配合 list-item 属性一起使用,通过设置 list-style-type 属性的值可以更改列表序号的外观,否则,列表序号将采用默认外观圆盘。

list-style-type 属性可取如下的值:

```
disc,circle,square,decimal,lower-roman,upper-roman,lower-alpha,upper-alpha,none
```

例如:

```
list-style-type: lower-roman
```

将列表项目符号的外观指定为小写罗马数字(i,ii,iii…)。以下是各个属性值的具体效果:

- disc 圆盘
- circle 圆圈
- square 方块
- decimal 十进制数
- lower-roman 小写罗马数字
- upper-roman 大写罗马数字
- lower-alpha 小写英文字母
- upper-alpha 大写英文字母

在下面的例 6 中,标记 chapter 和 section 的内容的显示方式都是列表方式,其中 section 是 chapter 的子标记。浏览器显示 example8_6.xml 的效果如图 8.7 所示。

【例 6】

sixCSS.css

```
book
{   display:block;
    background-color:rgb(229,227,226);
}
chapter
{   display:list-item;
    list-style-type:decimal;
    margin-left:25;
```

1. XML简介
   I. XML与HTML的区别
   II. XML解析器
2. 规范的XML文件
   I. XML声明
   II. XML中的标记

图 8.7 使用列表显示数据

```
    font - size:12pt;color:blue;
}
section
{   display:list - item;
    list - style - type:upper - roman;
    margin - left:35;
    font - size:10pt;color:black;
}
```

example8_6. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<?xml - stylesheet href = "sixCSS.css" type = "text/css" ?>
<book>
  <chapter>  XML 简介
     <section>XML 与 HTML 的区别</section>
     <section>XML 解析器</section>
  </chapter>
  <chapter>  规范的 XML 文件
     <section>XML 声明</section>
     <section>XML 中的标记</section>
  </chapter>
</book>
```

8.5.4 不显示

在样式表的样式规则中通过将属性 display 的值指定为 none，以达到不显示标记中的文本的目的。例如：

```
name
{  display:none;
}
```

对应 name 样式表的标记将不显示标记中的文本。

8.6 字 体

样式表中的“样式规则”是若干个用分号分隔的“属性名：属性值”，样式表中与字体有关的属性包括：

```
font - family, font - style, font - variant, font - weight, font - size
```

以下分别叙述上述属性的取值情况。

1. font-family 属性

font-family 属性的值是浏览器支持的字体名称，该属性的默认值是浏览器确定的默认字体名称，例如“宋体”。如果名称中有空格，属性值必须用双引号括起来，例如：

```
font - family:Arial;
font - family:宋体;
font - family:"Time New Roman";
```

2. font-style 属性

font-style 属性的值指定字体是否使用斜体，默认值是 normal。该属性值可以是 normal 或 italic，例如：

```
font - style:italic
font - style:normal
```

3. font-variant 属性

font-variant 属性的值用来指定是否使用小体的大写字母来显示文字，默认值是 normal。该属性能取的属性值有 normal(正常大写字母)和 small-caps(小体大写字母)，例如：

```
font - variant:small - caps
```

4. font-weight 属性

font-weight 属性的值用来设置字体线条的粗细，默认值是 normal。该属性能取的属性值如下：

```
normal,bold,bolder,lighter,100,200,300,400,500,600,700,800,900
```

例如：

```
font - weight: bold
font - weight: 600
```

属性值中的 normal 相当于 400，bold 相当于 700，bolder 相当于 500，lighter 相当于 300。

5. font-size 属性

font-size 属性的值用来设置字体的大小，单位为 pt(镑)。例如：

```
font - size:12pt
```

注意：如果子标记的样式表中不指定文本的字体属性，就会使用父标记的样式表中的字体属性及属性值。

下面的例 7 中，CSS 文件中的样式表使用了和字体有关的属性。用浏览器打开 example8_7.xml 的效果如图 8.8 所示。

【例 7】

sevenCSS. css

```
goods
{    display:block;
     font - family:楷体_GB2312;
     font - style:italic;
     font - weight:600;
```

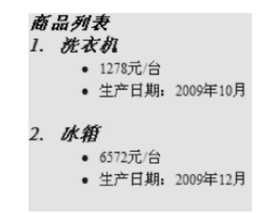


图 8.8 设置字体有关的属性值

```
    background - color:rgb(229,227,226);
}
name
{   display:list - item;
    list - style - type:decimal;
    margin - left:25;
    font - size:12pt;color:blue;
    font - family:黑体;
    font - weight:300;
}
price,madeTime
{   display:list - item;
    list - style - type:disc;
    margin - left:35;
    font - size:10pt;
    font - family:宋体;
    font - weight:300;
    font - style:normal
}
```

example8_7.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<?xml - stylesheet href = "sevenCSS.css" type = "text/css" ?>
<goods>商品列表
  <name>洗衣机
     <price>1278 元/台 </price>
     <madeTime>生产日期: 2009 年 10 月</madeTime>
  </name>
  <name>冰箱
     <price>6572 元/台 </price>
     <madeTime>生产日期: 2009 年 12 月</madeTime>
  </name>
</goods>
```

8.7 文本样式

样式表中与文本样式有关的属性包括：

```
text - align,text - indent,text - transform,text - decoration,vertical - align,line - height
```

以下分别叙述上述属性的取值情况。

1. text-align 属性

text-align 属性的值用来设置文本的对齐方式，其默认值是 left。该属性能取的属性值如下：

```
left(左对齐),right(右对齐),center(居中对齐),justify(两端对齐)
```

例如：

```
text - align:center
```

2. text-indent 属性

text-indent 属性的值用来设置文本首行的缩进量，单位是像素(px)或磅(pt)，默认值是 0。例如：

```
text - indent:16pt
```

3. text-transform 属性

text-transform 属性的值用来设置是否将文本中的字母全部大写、全部小写或者首字母大写，默认值是 none，该属性能取的属性值如下：

```
uppercase, lowercase, capitalize, none
```

例如：

```
text - transform:uppercase
```

4. text-decoration 属性

text-decoration 属性的值用来设置是否将文本加下画线，默认值是 none，该属性能取的属性值如下：

```
none, underline, overline, line - through, blink
```

例如：

```
text - decoration:underline
```

5. vertical-align 属性

vertical-align 属性的值用来设置文本的垂直对齐方式，默认值是 middle，该属性能取的属性值如下：

```
baseline, sub, super, top, text - top, middle, bottom, text - bottom
```

例如：

```
vertical - align:baseline
```

6. line-height 属性

line-height 属性的值用来设置文本的行距，默认值是 1，该属性的取值是正数。例如：

```
line - height:1.5
```

注意：如果子标记的样式表中不指定文本的样式属性，子标记就会使用父标记的样式表中的文本样式属性及属性值。

下面的例 8 中 CSS 文件中的样式表使用了和文本样式有关的属性，以便显示一个数学公式和两个化学分子式。用浏览器打开 example8_8.xml 的效果如图 8.9 所示。

【例 8】

eightCSS.css

```
math
{   display:block;
    font-size:12pt;
    font-style:italic;
    background-color:rgb(227,228,229);
}
chemistry
{   display:block;
    font-size:12pt;
    text-decoration:underline;
    background-color:cyan;
}
sup
{   display:line;
    font-size:8pt;
    font-style:italic;
    vertical-align:super;
}
low
{   display:line;
    font-size:8pt;
    vertical-align:bottom;
}
```

图 8.9 设置文本样式

example8_8.xml

```
<?xml  version="1.0"  encoding="UTF-8" ?>
<?xml-stylesheet href="eightCSS.css" type="text/css" ?>
<root>
    <math>
     平方和公式:(A+B)<sup>2</sup>=A<sup>2</sup>+2AB+B<sup>2</sup>
  </math>
  <chemistry>水的分子式: H<low>2</low>0</chemistry>
  <chemistry>二氧化碳的分子式:CO<low>2</low></chemistry>
</root>
```

8.8 边　　框

可以按文本的显示形式为文本添加边框。如果文本是按块(block)方式显示的,那么边框就是块的边框;如果文本是按行(line)方式显示的,那么边框就是行的边框;如果文本是按列表(list-item)方式显示的,那么边框就是列表的边框。

样式表中与文本边框有关的属性包括:

```
border-style,border-right-width,border-right-style,border-right-color
```

以下分别叙述上述属性的取值情况。

1. border-style 属性

该属性的默认值是 none,即文本默认没有边框。为了给文本增加边框,样式表中首先要设置 border-style 属性的值,使得文本具有边框,然后再设置其他属性的值。border-style 属性可取的值有:

```
none,dotted,dashed,solid,double,groove,ridge,inset,outset
```

例如:

```
border - style:dotted
```

为文本设置点状的边框。border-style 属性的各个属性值代表的边框效果如下:

- dotted　边框线是点组成的虚线。
- dashed　边框线是短线组成的虚线。
- double　边框线是双线。
- groove　3D 沟槽状边框。
- ridge　3D 脊状的边框。
- inset　3D 内嵌边框。
- outset　3D 外嵌边框。
- solid　普通的边框。

2. border-right-width 等属性

border-top-width、border-bottom-width、border-right-width 和 border-left-width 等属性用来设置边框的上边、底边、右边和左边的宽度,这些属性的默认值都是 1。可以重新设置这些属性的值来改变边框的上边、底边、右边和左边的宽度,例如:

```
border - top - width:12;
border - left - width:22;
```

3. border-right-style 等属性

border-right-style、border-left-style、border-bottom-style、border-top-style 等属性用来单独设置边框的右边、左边、底边和上边的样式。也就是说,在设置了 border-style 属性后,可以再单独设置这些属性,修改边框的某个边的样式,该 4 个属性的取值范围和 border-style 的相同。例如,border-style 属性的值是 dotted,那么边框的四个边的样式都是点状样式。如果想使底边的样式是虚线,就可进行如下的设置:

```
border - bottom - style:dashed
```

4. border-right-color 等属性

border-right-color、border-left-color、border-bottom-color、border-top-color 等属性用来设置边框的颜色,这些属性的默认值灰色。可以为这些属性重新设置颜色值,改变边框的颜色,例如:

```
border - right - color:blue
```

在下面的例 9 中,有 3 个标记的内容的显示方式都是块方式,其中的 2 个子标记的块区域被限制在父标记的块区域中。通过使用边框可明显地分辨出这 3 块区域。浏览器显示 example8_9.xml 的效果如图 8.10 所示。

【例 9】

nineCSS.css

```
Beijing
{    display:block;
     border-style:double;
     width=260;
     height=120
}
Haidian
{    display:block;
     border-style:dotted;
     width=150;
     height=60 ;
     font-size:10pt;
}
Xuanwu
{    display:block;
     border-style:ridge;
     width=90;
     height=30;
     font-size:10pt;
}
```


图 8.10 使用边框

example8_9.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet href="nineCSS.css" type="text/css" ?>
<China>
     <Beijing>北京市
      <Haidian>海淀区</Haidian>
      <Xuanwu>宣武区 </Xuanwu>
   </Beijing>
</China>
```

8.9 边　　缘

有时我们需要设置边缘,边缘是文本周围不可见的区域。如果文本是按块显示的,那么边缘就是块的边缘;如果文本是按行显示的,那么边缘就是行的边缘;如果文本是按列表显示的,那么边缘就是列表的边缘。

和边缘有关的属性包括:

```
margin-top,margin-right,margin-bottom,margin-left
```

例如：

```
margin - top:20; margin - right:120;margin - left:10;
```

分别设置了上边缘、右边缘和左边缘的大小，单位是像素。

在下面的例 10 中，有 2 个标记的内容的显示方式都是块方式，并设置了边缘的大小。浏览器显示 example8_10. xml 的效果如图 8.11 所示。

【例 10】

tenCSS. css

```
grade1
{
   display:block;
   border - style:ridge;
   border - top - width:15;
   margin - top:2;
   margin - left:2;
   margin - right:2;
   text - align:center;
   font - size:18pt;color:red;
   width = 230;
   height = 100;
}
name,authorized
{
  display:list - item;
  margin - left:22 ;
  text - align:left;
  font - size:12pt;color:green;
}
grade2
{
   display:block;
   border - style:dotted;
   border - top - width:10;
   margin - top:12;
   margin - left:2 ;
   margin - right:2;
   text - align:center;
   font - size:18pt;color:blue;
   width = 230;
   height = 100;
}
```

图 8.11 设置边缘

example8_10. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<?xml - stylesheet href = "tenCSS.css" type = "text/css" ?>
< root >
```

```
    <grade1>
     高级灌水师
     <name> 姓名: 小黑 </name>
     <authorized> 发证单位: 黑魔方网站 </authorized>
  </grade1>
  <grade2>
     初级灌水师
     <name> 姓名: Nancy </name>
     <authorized> 发证单位: 黑魔方网站 </authorized>
  </grade2>
</root>
```

8.10 颜色和背景

如果需要设置文本的字符颜色和背景颜色,就可以使用属性 color 和 background-color。

通过设置 color 属性的值可以指定文本的字符颜色,例如:

```
color:rgb(120,225,220);
color:red;
```

通过设置 background-color 属性的值可以指定文本的背景颜色,例如:

```
background-color:rgb(10,225,220);
background-color:yellow;
```

8.11 显示图像

可以通过设置 background-image 属性的值为文本指定一幅背景图像。background-image 属性的取值形式为:

```
URL("文件名字");
```

例如:

```
background-image:URL("animal.jpg");
```

另外,还可以通过 background-repeat 属性设置图像是否通过重复出现来平铺背景。background-repeat 取值如下:

```
repeat,repeat-x,repeat-y,no-repeat
```

例如:

```
background-repeat:repeat
```

其中 repeat-x 和 repeat-y 表示仅仅沿水平或垂直方向重复图像。

也可以单独地显示一幅图像,并指定图像和周围文本的位置关系。为了单独显示一幅图像,只要形式上为一个空标记指定一幅背景图像即可。由于空标记没有文本内容,其背景的大小为 0×0,所以相关的样式表必须将 display 设置为 block,并使用 width 和 height 属性指定块区域的大小。

下面的例 11 除了给一个标记的文本设置背景图像外,还通过形式上为一个空标记设置背景,将一幅图像单独显示出来,效果如图 8.12 所示。

【例 11】

elevenCSS. css

```
tom
{
   display:block;
   position:absolution;
   top = 50;
   left = 10;
   width = 150px;
   height = 120px;
   text - align:center;
   font - size:18pt;color:blue;
   font - weight:bold;
   background - image:URL(tom. jpg);
   background - repeat:no - repeat
}
image
{
   display:block;
   position:absolution;
   top = 10;
   left = 180;
   width = 150px;
   height = 200px;
   background - image:URL(flower. gif);
   background - repeat:repeat
}
```



图 8.12　显示图像

example8_11. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<?xml - stylesheet href = "elevenCSS. css" type = "text/css" ?>
< root >
    < tom >
    俺是著名的 TOM 老猫,很多人使用俺的形象
  </tom >
  < image/>
</root >
```

8.12 设置鼠标的形状

如果希望控制鼠标指针运动到文字的显示区域上面时的形状，就可以使用 cursor 属性。cursor 属性可以取的值有：

auto,crosshair,default,hand,move,e-resize,ne-resize,nw-resize,n-resize,se-resize,sw-resize,s-resize,w-resize,text,wait,help

例如：

```
cursor:hand;
```

指定鼠标指针运动到文字的显示区域上面时变成"手"的形状。

例 12 中，两个标记的显示方式是块区域，当鼠标指针在块区域上方时改变形状。

【例 12】

twelveCSS.css

```
mouse#A1
{   display:block;
    cursor:hand;
    font-size:16pt;
    color:red;
    width=130;
    height=100;
    border-style:double;
}
mouse#A2
{   display:block;
    cursor:move;
    font-size:16pt;
    color:blue;
    width=130;
    height=100;
    border-style:double;
}
```

example8_12.xml

```
<?xml  version="1.0"  encoding="UTF-8" ?>
<?xml-stylesheet href="twelveCSS.css" type="text/css" ?>
<root>
      <mouse ID="A1">
      鼠标在我上面是手的形状
   </mouse>
   <mouse ID="A2">
      鼠标在我上面是运动的形状
   </mouse>
</root>
```

8.13 处理层叠

样式表通过指定属性 display 的值来设置文本的显示方式,例如,display:block 使得文本将在浏览器的一个块区域中显示。样式表中还可以通过设置 position、width 和 height 属性的值来准确地设置显示区域的位置和大小,那么样式表设置的显示区域就有可能发生重叠。

可以在样式表中设置 z-index 属性的值来规定一个样式表所在的层,z-index 属性的值应是正整数,称为样式表的层数。当样式表之间的显示区域发生重叠时,具有较大层数的样式表的显示区域将遮挡具有较小层数的样式表的显示区域。

注意:如果不设置 z-index 属性的值,后被使用的样式表的显示区域将遮挡先被使用的样式表的显示区域。

在下面的例 13 中,两个样式表的显示区域发生了重叠,浏览器显示 example8_13.xml 的效果如图 8.13 所示。

【例 13】

thirteenCSS.css

```
house
{    display:block;
     position:absolute;
     top=6;
     left=10;
     width=150px;
     height=120px;
     background-image:URL("house.jpg");
     background-repeat:no-repeat;
     z-index:10;
}
road
{    display:block;
     position:absolute;
     top=6;
     left=110;
     width=170px;
     height=110px;
     background-image:URL("road.jpg");
     background-repeat:no-repeat;
     z-index:5;
}
```



图 8.13 处理层叠

example8_13.xml

```
<?xml  version="1.0"  encoding="UTF-8" ?>
<?xml-stylesheet href="thirteenCSS.css" type="text/css" ?>
<city>
```

```
    <road/>
    <house/>
</city>
```

习 题 8

1. XML 文件使用什么指令来使用 CSS?

2. 若 XML 有一个标记的名字是“学生”,要使该标记中的文本内容在一个块区域中显示、字体的颜色是“红色”、字体的大小是 18 磅,对应的 CSS 应当提供怎样的样式表?

3. 若 XML 有三个标记的名字都是“学生”,都有“ID”属性。请编写 CSS 使得三个该标记中的文本内容分别在块区域中显示,要求三个块区域的边框互不相同。

4. 有下列 XML 文件以及 CSS 层叠样式表,若需显示效果如图 8.14 所示的数据,应当修改 XML 文件还是 CSS 层叠样式表文件?

Xiti4.xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<?xml - stylesheet href = "showStudent.css" type = "text/css" ?>
<root>
    <student>
      王开出
      <sex>男 </sex>
      <birthday>1970.12.28 出生 </birthday>
      <address> 广东 </address>
    </student>
</root>
```

showStudent.css

```
student
{  display:block;
   display:list - item;
   list - style - type:decimal;
   margin - left:30;
   font - size:10pt;
   color:black;
}
sex
{  display:list - item;
   list - style - type:lower - roman;
   margin - left:60;
   font - size:8pt;
   color:green;
}
birthday
{  display:list - item;
   list - style - type:lower - roman;
```

1. 张三
    i. 男
    ii. 1990.7.8 出生
    iii. 北京
2. 李四
    i. 女
    ii. 1988.12.24 出生
    iii. 天津
3. 孙五
    i. 男
    ii. 1989.8.26 出生
    iii. 上海

图 8.14 用样式表显示数据

```
    margin-left:60;
    font-size:8pt;
    color:blue;
}
address
{   display:list-item;
    list-style-type:lower-roman;
    margin-left:60;
    font-size:8pt;
    color:pink;
}
```

5. 请为下列 XML 文件编写相应的 CSS,使得 XML 文件的数据显示效果如图 8.15 所示。

Xiti5.xml

```
<?xml  version="1.0"  encoding="UTF-8" ?>
<?xml-stylesheet href="show.css" type="text/css" ?>
<root>
    <math>
      S<low>n</low>=
       a<low>1</low>+a<low>2</low>+ … +a<low>n</low>
  </math>
  <chemistry>
    (NH<low>4</low>)<low>2</low>0,
      CO<low>2</low>.
  </chemistry>
</root>
```

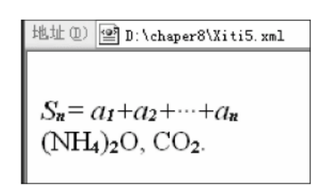


图 8.15 样式表显示数据的效果

第9章 XML Schema模式简介

主要内容

- 什么是 XML Schema
- XML Schema 中的标记
- XML Schema 模式的验证
- 简单类型元素
- 复杂类型元素
- 属性

XML Schema 模式的内容非常庞大，详细地讲解 XML Schema 的内容已经超出了本书的范围，本书就 XML Schema 最基本的内容给予简单的介绍，读者可以访问

www.w3.org/TR/xmlschema-1/.XML

了解有关最新的进展情况。XML Schema 的标准稍显宽泛了一些，如果对 XML 文件的约束只限于文件的标记和属性结构，而不涉及文本的具体内容，那么使用 DTD 即可。DTD 能够完成 XML Schema 模式的大部分功能，且简单、兼容性更好，第 4 章和第 5 章学习的 DOM 解析器和 SAX 解析器都能用来验证 XML 文件是否遵守了 DTD 模式的约束。

9.1 什么是 XML Schema

XML 标记的内容可以由文本数据和标记组成，模式就是为了限制标记应当有怎样的文本内容以及可以有哪些子标记。

第 3 章介绍了 DTD 文件，及如何使用 DTD 文件约束 XML 文件。可以将 DTD 文件看做 XML 文件的一种模式，一个和 DTD 关联的有效的 XML 文件必须遵守该模式。但是，DTD 文件也有不足之处。例如，当 DTD 使用 Element 元素将一个标记约束为“#PCDATA”时，仅仅是限制了该标记只能有文本数据，却不能限制文本数据的实际意义。再如，不能强制限制文本内容是浮点数或是日期形式的数据，例如“28.16”或“2010-12-26”。

W3C XML Schema 给出了一种新的模式，简称 XML Schema 模式，该模式不仅能实现 DTD 的大部分功能，而且能指定标记内容的“数据类型”。但 XML Schema 模式也不

是万能的,XML Schema 模式的出现并不意味着抛弃 DTD,DTD 可以实现 XML Schema 模式不能实现的功能,而且较 XML Schema 模式而言,具有更广泛的解析器支持。

DTD 非常适合下列情形:

① 文件是叙述性的,并有混合内容。

② 需要约束标记之间的关系,特别是子标记的顺序关系,而不是标记本身的文本内容。

③ 需要使用实体。

④ XML 文件的使用者对使用的 DTD 达成一致。

XML Schema 非常适合下列情形:

① 需要定义数据类型,以便约束标记的文本内容的结构,可以约束“日期”标记的内容是“yyyy-mm-dd”格式的字符串,如“2010-12-12”。

② 标记的子标记的顺序并不重要,重要的是其数量。

③ 标记约束不限于父子关系,也可以是祖先及子孙关系。

④ 跨越多个文件,名称空间前缀不一致。

相对与 DTD,XML Schema 模式的最大优势就是可以约束 XML 标记的数据类型。若有一个 XML 文件,根标记是“商品列表”,要求根标记有若干个子标记“商品”,要求每个“商品”标记顺序地有“名称”、“生产日期”和“价格”子标记。如果非常关心标记“生产日期”和“价格”的数据的表示形式,希望约束“生产日期”标记的内容必须是“yyyy-:mm-dd”形式、希望约束“价格”标记的内容必须是数字形式的序列,那么 DTD 无能为力。而 XML Schema 模式可以约束标记的数据类型,这里所说的数据类型是指数据的表示形式。例如,一个标记的内容被约束为 int 型,那么该标记的文本内容必须是由数字型字符组成。

9.2 XML Schema 中的标记

XML Schema 模式是扩展名为“.xsd”的一个文本文件,使用 XML 语法来编写,这一点和 DTD 文件不同,保存时所选择的编码必须和所约束的 XML 文件一致。例如,XML Schema 所要约束的 XML 文件的编码为 UTF-8,那么 XML Schema 模式也必须按照 UTF-8 编码保存。若在保存文件时,系统总是自动在文件名尾加上“.txt”,那么在保存文件时可以将文件名用双引号括起来。

9.2.1 根标记

XML Schema 模式的根标记必须是 schema,使用的名称空间必须是:

```
http://www.w3.org/2001/XMLSchema
```

名称空间的前缀是 xsd。例如:

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  ⋮
```

```
</xsd:schema>
```

9.2.2 元素标记

XML Schema 模式的主要目的是用 element 标记来约束 XML 文件中的标记。可以将 element 标记作为 XML Schema 模式中根标记的子标记来使用，XML Schema 模式中的 element 标记简称为元素。若元素是根标记的子标记，则称为全局元素，全局元素的作用是约束 XML 文件中任何级别上的子标记，无论该 XML 标记是 XML 文件中的哪一级子标记。以下详细讲解有关 element 标记的使用细节。

对于 XML 文件中没有子标记的标记，XML Schema 模式使用“简单类型”元素来约束。XML Schema 中“简单类型”元素的格式为：

```
<xsd:element name = "标记名称" type = "简单数据类型"/>
```

其中，“标记名称”就是对应的 XML 文件中标记的名称，“简单数据类型”对标记中文本数据进行限制。例如，XML Schema 模式有如下的元素：

```
<xsd:element name = "生产日期"type = "xsd:date"/>
```

那么使用该模式进行约束的 XML 文件中的任何名字为“生产日期”的标记中的文本数据必须是日期类型。

XML Schema 模式可以使用的简单数据类型有：int、float、double、date、time、string 等，例如元素：

```
<xsd:element name = "价格" type = "xsd:float"/>
```

约束 XML 文件中的“价格”没有子标记，且标记的数据必须是浮点数据，即内容必须是数字(可以有小数点)组成的序列，例如 6785.89、1278.66 等。下列元素：

```
<xsd:element name = "开车时间" type = "xsd:time"/>
```

约束 XML 文件中的“开车时间”没有子标记，且标记的数据必须是时间，即内容必须是“hh:mm:ss”形式，例如，08:26:00。

对于 XML 文件中有子标记的标记，XML Schema 模式文件使用“复杂类型”元素来约束。XML Schema 中“复杂类型”元素的格式为：

```
<xsd:element name = "标记名称" >
  <xsd:complexType>
      ⋮
   </xsd:complexType>
</xsd:element>
```

以下一个“复杂类型”元素：

```
<xsd:element name = "商品" >
```

```
        <xsd:complexType>
          <xsd:sequence>
              <xsd:element  ref="名称"/>
              <xsd:element  ref="生产日期"/>
              <xsd:element  ref="价格"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
```

上述元素约束任何名字为“商品”的标记必须顺序地有三个名字分别为“名称”、“生产日期”和“价格”的子标记,其中

```
    <xsd:element  ref="子标记名字"/>
```

的作用是对“子标记名字”指定的 XML 子标记的约束引用 Schema 模式中全局元素给出的约束条件。

下面是一个约束 XML 文件中名字为“商品”标记的“复杂类型”元素:

```
    <xsd:element name="商品">
        <xsd:complexType>
          <xsd:all>
              <xsd:element  ref="名称"/>
              <xsd:element  ref="生产日期"/>
              <xsd:element  ref="价格"/>
            </xsd:all>
        </xsd:complexType>
    </xsd:element>
```

上述 element 元素约束任何名字为“商品”的标记必须有三个名字分别为“名称”、“生产日期”和“价格”的子标记,但三个子标记的顺序可任意排列,不受限制。

9.2.3 属性标记

对于 XML 文件中的属性,XML Schema 模式使用 attribute 标记来给予约束,我们在后续的 9.6 节讲解 attribute 标记的使用细节。

9.3 XML Schema 模式的验证

XML Schema 模式的目的是约束一个规范的 XML 文件,也称一个受某个 XML Schema 模式约束的 XML 文件是有效的 XML 文件。

显然,人们不愿意人工地来验证 XML 文件是否遵守了 XML Schema 模式所规定的约束条件,则可以使用 Sun 公司 SDK1.5 后续版本提供的 API 验证一个 XML 文件是否遵守了某个 XML Schema 模式。

验证一个 XML 文件是否遵守了某个 XML Schema 模式的步骤如下。

1. 得到一个 SchemaFactory 对象

使用 SchemaFactory 类的静态方法

```
static SchemaFactory newInstance(String schemaLanguage)
```

得到一个 SchemaFactory 对象,该方法中参数的取值必须是:

```
"http://www.w3.org/2001/XMLSchema"
```

例如:

```
SchemaFactory schemaFactory =
SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
```

2. 创建 Schema 对象

步骤 1 中得到的 SchemaFactory 对象调用

```
Schema newSchema(File schema)
```

方法,可以返回一个 Schema 对象,例如:

```
Schema schema = schemaFactory.newSchema(new File("pattern.xsd"));
```

3. 得到验证器

步骤 2 中得到的 Schema 对象调用

```
Validator newValidator()
```

方法返回一个验证器,例如:

```
Validator validator = schema.newValidator();
```

4. 使用验证器

验证器首先调用

```
void setErrorHandler(ErrorHandler errorHandler)
```

方法设置负责报告错误的处理器,其中参数取值必须是实现 ErrorHandler 类的实例。DefaultHandler 类是实现了 ContentHandler、DTDHandler、EntityResolver 和 ErrorHandler 接口的类,可以用 DefaultHandler 类的子类的实例作为报告错误的处理器。

然后再调用

```
public void validate(Source source)
```

方法验证 XML 文件是否遵守了 XML Schema 模式,例如:

```
validator.validate(new StreamSource(new File("example10_1.xml"))) ;
```

在下面的例 1 中,有一个简单的 XML Schema 模式文件 schema9_1.xsd,用来约束 example9_1.xml 文件。要求"商品列表"可以有若干个"商品"标记,每个"商品"标记必须有顺序地有"名称"、"生产日期"和"价格"子标记,要求标记"生产日期"的内容必须是

“yyyy-mm-dd”形式，标记“价格”的内容必须是浮点数形式。使用例 1 的 TestSchema. java 文件进行验证的效果如图 9.1 所示。

XML文件:example9_1.xml符合模式

图 9.1 使用 Schema 模式验证 XML 文件

【例 1】

example9_1.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<商品列表>
   <商品>
      <名称>电视机</名称>
      <生产日期> 2010 - 12 - 12 </生产日期>
      <价格> 5673.89 </价格>
   </商品>
   <商品>
      <名称>洗衣机</名称>
      <生产日期> 2010 - 10 - 10 </生产日期>
      <价格> 3673.67 </价格>
   </商品>
</商品列表>
```

schema9_1.xsd

```
< xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  < xsd:element name = "商品列表" > <! -- 对根标记的约束 -->
    < xsd:complexType >
      < xsd:sequence >
        < xsd:element  ref = "商品" minOccurs = "1" maxOccurs = "unbounded" />
      </xsd:sequence >
    </xsd:complexType >
  </xsd:element >
  < xsd:element name = "商品" >    <! -- 对商品标记的约束 -->
    < xsd:complexType >
      < xsd:sequence >
        < xsd:element ref = "名称"/>
        < xsd:element ref = "生产日期"/>
        < xsd:element ref = "价格"/>
      </xsd:sequence >
    </xsd:complexType >
  </xsd:element >
  < xsd:element name = "名称" type = "xsd:string"/>  <! -- 对名称标记的约束 -->
  < xsd:element name = "生产日期" type = "xsd:date"/>  <! -- 对生产日期标记的约束 -->
  < xsd:element name = "价格" type = "xsd:float"/>  <! -- 对价格标记的约束 -->
</xsd:schema >
```

TestSchema.java

```
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.*;
import java.io.*;
import javax.xml.validation.*;
import javax.xml.transform.stream.StreamSource;
public class TestSchema{
   public static void main(String args[]){
      File xsdfile = new File("schema9_1.xsd");
      File xmlfile = new File("example9_1.xml");
      Handler errorHandler = null;
      try{ SchemaFactory schemaFactory =
           SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
           Schema schema = schemaFactory.newSchema(xsdfile);
           Validator validator = schema.newValidator();
           errorHandler = new Handler();
           validator.setErrorHandler(errorHandler);
           validator.validate(new StreamSource(xmlfile)) ;
       }
       catch(Exception e){
           System.out.println(e);
        }
       if(errorHandler.errorMessage == null)
           System.out.println("XML 文件:" + xmlfile.getName() + "符合模式");
       else
           System.out.println("XML 文件:" + xmlfile.getName() + "不符合模式");
   }
}
class Handler extends DefaultHandler{
   String errorMessage = null;
   public void error(SAXParseException e) throws SAXException{
       errorMessage = e.getMessage();
       int row = e.getLineNumber();
       int colums = e.getColumnNumber();
       System.out.println("一般错误: " + errorMessage + "位置: " + row + "," + colums);
   }
   public void fatalError(SAXParseException e) throws SAXException{
       errorMessage = e.getMessage();
       int row = e.getLineNumber();
       int colums = e.getColumnNumber();
       System.out.println("致命错误: " + errorMessage + "位置: " + row + "," + colums);
    }
}
```

如果将 example9_1.xml 中“电视机”的生产日期更改为“2010-04-31”,那么将导致不符合模式的错误,因为 4 月没有 31 日,图 9.2 所示为程序检查出的错误信息。

一般错误：cvc-datatype-valid.1.2.1: '2010-04-31' is not a valid value
.位置：5,30

图 9.2 检查出的错误

9.4 简单类型元素

对于 XML 文件中没有子标记的标记，即只有文本数据的标记，XML Schema 模式可以使用内建的"简单类型"元素来给予约束。XML Schema 中内建的"简单类型"元素的格式为：

```
<xsd:element name = "标记名称" type = "简单数据类型"/>
```

如果内建的"简单类型"元素是全局元素，即是"schema"根标记的子标记，那么内建的"简单类型"元素可以约束由该元素 name 属性值指定的 XML 文件中的 XML 标记，无论该 XML 标记是 XML 文件的哪一级子标记。内建"简单类型"元素标记中涉及 type 属性的值，正是这个值决定了该 element 元素标记是内建的"简单类型"元素。对于内建的"简单类型"元素，type 属性的取值是 XML W3C 标准规定的简单数据类型。表 9.1 列出了常用的 XML W3C 标准规定的简单数据类型。

表 9.1 内建的简单数据类型

类型	描述
byte	占 1 个字节的整数
short	占 2 个字节的整数
int	占 4 个字节的整数
long	占 8 个字节的整数
float	单精度浮点数
double	双精度浮点数
string	字符串数据
date	用 yyyy-mm-dd 形式表示的日期
time	用 hh:mm:ss 形式表示的时间
boolean	true,1,false,0 四个值
integer	整数，不限制大小
negativeInteger	负整数，不限制大小
nonNegativeInteger	非负整数，不限制大小
positiveInteger	正整数，不限制大小
nonPositiveInteger	非正整数，不限制大小
unsignedByte	占 1 个字节的无符号整数
unsignedShort	占 2 个字节的无符号整数
unsignedInt	占 4 个字节的无符号整数
unsignedLong	占 8 个字节的无符号整数

在下面的例 2 中，XML 文件 example9_2. xml 中的“列车”标记有三个子标记：“车次”、“始发时间”和“车厢数目”，要求 XML 文件符合 schema9_2. xsd 模式，“名称”是 string 类型数据，“始发时间”是 time 类型数据，“车厢数目”是 positiveInteger 类型数据。

【例 2】

example9_2. xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<列车列表>
   <列车>
      <车次> T89 次</车次>
      <始发时间> 12:56:00 </始发时间>
      <车厢数目> 18 </车厢数目>
   </列车>
   <列车>
      <车次> K37 </车次>
      <始发时间> 22:12:00 </始发时间>
      <车厢数目> 16 </车厢数目>
   </列车>
</列车列表>
```

schema9_2. xsd

```
< xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  < xsd:element name = "列车列表" > <! -- 对根标记的约束 -->
    < xsd:complexType >
      < xsd: sequence >
        < xsd:element ref = "列车" minOccurs = "1" maxOccurs = "12" />
      </xsd:sequence >
    </xsd:complexType >
  </xsd:element >
  < xsd:element name = "列车" > <! -- 对列车标记的约束 -->
    < xsd:complexType >
      < xsd: sequence >
        < xsd:element ref = "车次"/>
        < xsd:element ref = "始发时间"/>
        < xsd:element ref = "车厢数目"/>
      </xsd:sequence >
    </xsd:complexType >
  </xsd:element >
  < xsd:element name = "车次" type = "xsd:string"/>   <! -- 对车次标记的约束 -->
  < xsd:element name = "始发时间" type = "xsd:tine"/> <! -- 对始发时间标记的约束 -->
  < xsd:element name = "车厢数目" type = "xsd:positiveInteger"/><! -- 对车厢数目标记的约束 -->
</xsd:schema >
```

如果将例 2example9_2. xml 中某个“车厢数目标记中的数据更改为－2；将某个“始发时间”更改为 11:61:10，将导致不符合模式的错误。用 9.3 节例 1 提供的 Java 程序进行验证(需要简单修改 Java 文件中模式文件和 XML 文件的名字)，将检查出这些错误。

Schema 模式中的简单元素可以通过指定 fixed 属性的值约束 XML 标记包含的文本内容必须是 fixed 属性的值,例如:

```
<xsd:element  = "报警电话" type = xsd:string  fixed = "110" />
```

约束名字是“报警电话”的 XML 标记包含的文本内容必须是“110”。

9.5 复杂类型元素

对于 XML 文件中有子标记的标记,XML Schema 模式可以使用“复杂类型”元素来给予约束。“复杂类型”元素中最重要的部分就是“对 XML 子标记约束的元素”部分,如果该“复杂类型”元素想约束 name 属性指定的标记顺序地出现几个子标记,那么就可以使用 sequence 子标记,并且在 sequence 子标记中引用全局元素(使用 ref 属性)对 XML 子标记进行约束,格式如下:

```
<xsd:element name = "标记名称" >
      <xsd:complexType>
        <xsd:sequence>
             <xsd:element  ref = "子标记 1" />
             <xsd:element  ref = "子标记 2" />
               ⋮
             <xsd:element  ref = "子标记 2" />
        </xsd:sequence>
      </xsd:complexType>
</xsd:element>
```

带 ref 属性的元素作用是。指定当前 element 元素约束的标记的子标记的名称,对该子标记的约束引用全局元素。

如果该“复杂类型”元素想约束 name 属性指定的标记出现几个子标记,但不约束它们出现的先后顺序,那么就可以使用 all 子标记,并且在 all 子标记中引用全局元素(使用 ref 属性)对 XML 子标记进行约束,格式如下:

```
<xsd:element name = "标记名称" >
      <xsd:complexType>
        <xsd:all>
             <xsd:element  ref = "子标记 1" />
             <xsd:element  ref = "子标记 2" />
               ⋮
             <xsd:element  ref = "子标记 2" />
        </xsd:all>
      </xsd:complexType>
</xsd:element>
```

注意:如果约束 XML 标记只出现一个子标记,则在 Schema 模式的“复杂类型”元素中可以省略 sequence 子标记或 all 子标记。

如果对子标记的约束不想引用全局元素，或没有约束该子标记的全局元素，对子标记约束的元素还可以是一个“复杂类型”元素。例如：

```
<xsd:element name = "标记名称" >
 <xsd:complexType>
  <xsd:sequence>
    <xsd:element name = "子标记 1" >
        <xsd:complexType >
           ⋮
        </xsd:complexType>
    </xsd:element>
    <xsd:element name = "子标记 2" >
        <xsd:complexType >
       ⋮
        </xsd:complexType>
     </xsd:element >
   </xsd:sequence>
 </xsd:complexType>
</xsd:element >
```

在下面的例 3 中，模式使用“复杂类型”元素约束 XML 文件中的标记。

【例 3】

example9_3. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<产品>
   <电视机 >
       <价钱> 5678 </价钱>
       <重量> 23.87 </重量>
  </电视机>
  <洗衣机 >
       <价钱> 5976 </价钱>
       <重量> 34 </重量>
  </洗衣机>
</产品>
```

schema9_3. xsd

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "产品" >
    <xsd:complexType>
      <xsd:sequence >
         <xsd:element  ref = "电视机" />
         <xsd:element name = "洗衣机" >
           <xsd:complexType>
             <xsd:sequence >
                 <xsd:element  ref = "价钱" />
                 <xsd:element name = "重量" type = "xsd:int"/>
             </xsd:sequence>
           </xsd:complexType >
         </xsd:element>
     </xsd:sequence >
```

```
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="电视机">
      <xsd:complexType>
        <xsd:sequence>
            <xsd:element  ref="价钱" />
            <xsd:element  ref="重量"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="价钱" type="xsd:int"/>
    <xsd:element name="重量" type="xsd:float"/>
</xsd:schema>
```

“复杂类型”元素的特殊情况能代替“简单类型”元素。例如,对于“简单类型”元素:

```
<xsd:element name="姓名" type= "xsd:string">
```

下列“复杂类型”元素的作用和它相同:

```
<xsd:element name="姓名">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="xsd:string" />
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

Schema 模式中的“复杂类型”元素在约束子标记出现的次数时,通过指定 minOccurs 属性或 maxOccurs 属性的值约束 XML 标记出现的次数,例如:

```
<xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="列车" minOccurs="1" maxOccurs="12" />
      </xsd:sequence>
</xsd:complexType>
```

约束名字是“列车”的 XML 标记最少要出现 1 次,最多可出现 12 次。而

```
<xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="列车" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
</xsd:complexType>
```

约束名字是“列车”的 XML 标记可以不出现次数,如果出现,出现的次数没有限制。

9.6 属　　性

对于 XML 文件中的属性,XML Schema 模式使用“attribute”标记来约束,该标记的格式如下:

```
<xsd:attribute name = "属性名字" type = "基本数据类型" use = "条件" />
```

其中 use 可取值 required、optional、fixed、default。

attribute 标记必须在“复杂类型”元素中使用，指出“复杂类型”元素约束的 XML 标记应当有怎样的属性。例如，对于：

```
<xsd:element name = "姓名" >
      <xsd:complexType>
          <xsd:simpleContent>
             <xsd:extension base = "xsd:string" >
                 <xsd:attribute name = "学号" type = "xsd:int" use = "required" />
             </xsd:extension>
          </xsd:simpleContent>
      </xsd:complexType>
  </xsd:element>
```

该“复杂类型”元素中使用了 attribute 标记，约束“姓名”标记必须要有“学号”属性。

在下面的例 4 中，模式使用 attribute 标记约束 XML 标记的属性。

【例 4】

example9_4. xml

```
<?xml  version = "1.0"  encoding = "UTF - 8" ?>
<产品信息>
   <产品 grade = "1">
        <名称 语言 = "汉语"> 电视机  </名称>
          <价钱 单位 = "人民币"> 5237 </价钱>
   </产品>
  <产品 grade = "2">
          <名称 语言 = "英语"> Radio </名称>
          <价钱 单位 = "欧元"> 176  </价钱>
   </产品>
</产品信息>
```

schema9_4. xsd

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "产品信息" >
    <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref = "产品" minOccurs = "1" maxOccurs = "8" />
       </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "产品" >
     <xsd:complexType>
         <xsd:sequence>
             <xsd:element ref = "名称" />
             <xsd:element ref = "价钱" />
         </xsd:sequence>
```

```
                <xsd:attribute name="grade" type="xsd:int" use="required" />
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="名称">
            <xsd:complexType>
                <xsd:simpleContent>
                    <xsd:extension base="xsd:string">
                        <xsd:attribute name="语言" type="xsd:string" use="required" />
                    </xsd:extension>
                </xsd:simpleContent>
            </xsd:complexType>
    </xsd:element>
    <xsd:element name="价钱">
            <xsd:complexType>
                <xsd:simpleContent>
                    <xsd:extension base="xsd:int">
                        <xsd:attribute name="单位" type="xsd:string" use="required" />
                    </xsd:extension>
                </xsd:simpleContent>
            </xsd:complexType>
    </xsd:element>
    </xsd:schema>
```

如果将 example9_4.xml 中某个“grade”属性的属性值更改为“一级”，将导致不符合模式的错误，因为 grade 属性被模式约束的类型是 int；如果将“example9_4.xm 中某个“名称”的“语言”属性省略，也将导致不符合模式的错误，因为“语言”属性被模式约束为必须要有的。用 9.3 节例 1 提供的 Java 程序进行验证(需要简单修改 Java 文件中模式文件和 XML 文件的名字)，将检查出这些错误。

习 题 9

1. 将例 1 中“生产日期”的数据写成 1996-04-23 是否可以？
2. 将例 2 中“始发时间”的数据写成 9:12:23 是否可以？
3. 编写一个关于职员工资信息的 XML 文件，使用模式约束“月薪”标记必须是 float 型数据。